COLLÈGE DE MAISONNEUVE

Notes de cours

PL/SQL (Oracle 11g)

Hakim Benameurlaine

Décembre 2012

Table of Contents

NOTE IMPORTANTE

Rapporter des erreurs ou des suggestions est une façon très utile de m'aider à améliorer cette documentation.

Merci d'avance

Hakim Benameurlaine

hbenameurlaine@cmaisonneuve.qc.ca

1 PRÉSENTATION DU PL/SQL

Le langage **PL/SQL** permet de normaliser le développement d'applications liées aux **B**ases de **D**onnées. C'est une extension du langage **SQL** développé par **IBM** (International Business Machine) dans les années **1970**. C'est un langage procédural par opposition à SQL qui est un langage déclaratif.

1.1 Bloc PL/SQL

Un bloc PL/SQL typique est constitué par les trois partie suivantes :

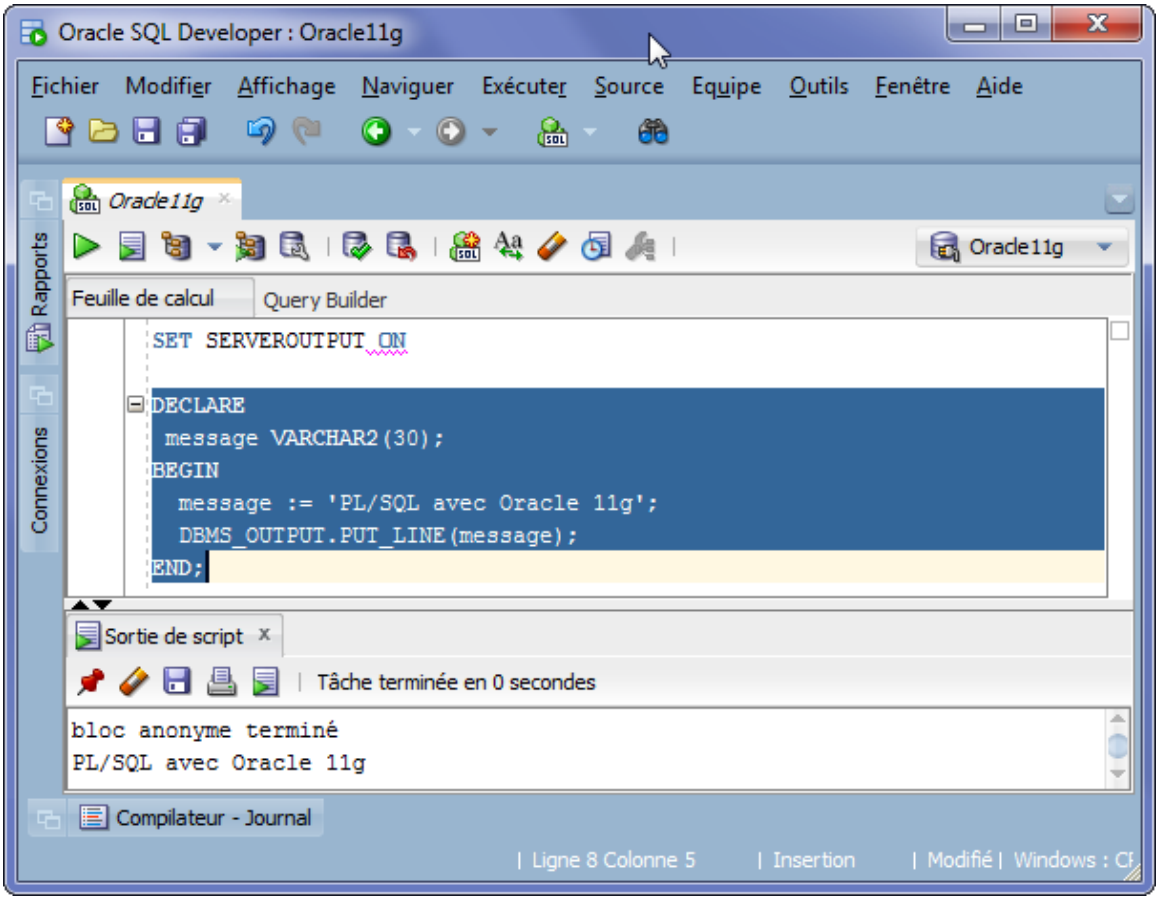
- Partie DECLARE dans laquelle on déclare les variables.
- Partie traitement entre BEGIN et END.
- Partie EXCEPTION pour le traitement des exceptions.

```
DECLARE
--Déclaration des variables (partie optionnelle)
BEGIN
--Code PL/SQL (si aucun code alors mettre null;)
EXCEPTION
--Traitement des exceptions (partie optionnelle)
END;
```

Voici un bloc PL/SQL minimal :

```
BEGIN
  NULL;
END
```

EXEMPLE

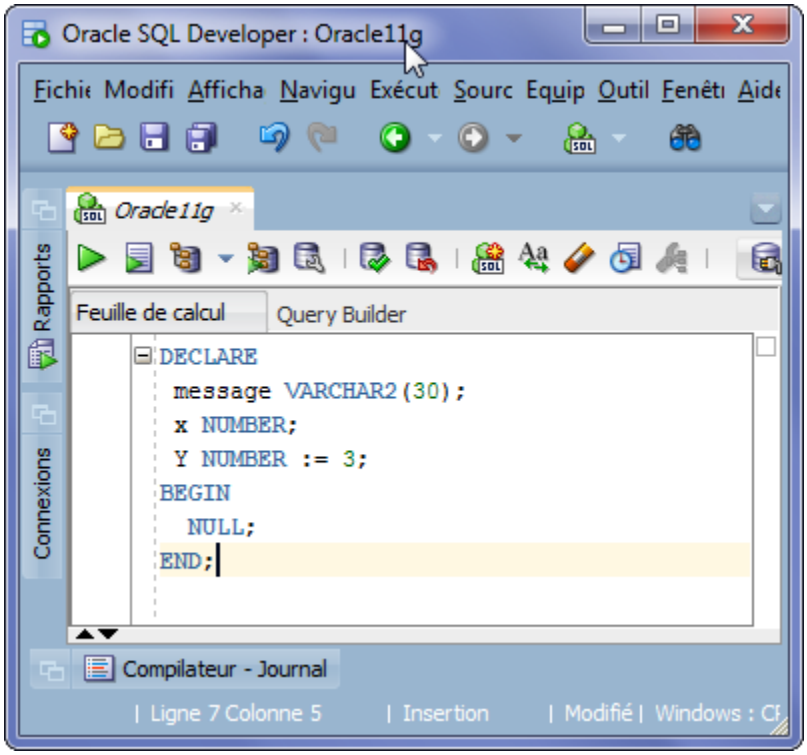

Oracle SQL Developer : Oracle11g
Fichier Modifier Affichage Naviguer Exécuter Source Equipe Outils Fenêtre Aide
Oracle11g
Rapports
Connexions
Oracle11g
Feuille de calcul
Query Builder
SET SERVEROUTPUT ON
DECLARE
message VARCHAR2(30);
BEGIN
message := 'PL/SQL avec Oracle 11g';
DBMS_OUTPUT.PUT_LINE(message);
END;
Sortie de script
Tâche terminée en 0 secondes
bloc anonyme terminé
PL/SQL avec Oracle 11g
Compilateur - Journal
Ligne 8 Colonne 5
Insertion
Modifié
Windows : Cf

1.2 Les variables

Éléments permettant de stocker une valeur en vue de son utilisation future.

- une variable locale est définie par l'utilisateur par l'instruction **DECLARE**,
- se voit affecter une valeur initiale par l'opérateur d'assignation **:=** ,
- elle est utilisée dans l'instruction ou le lot.

Il possible de déclarer une variable et lui assigner une valeur sur la même ligne.

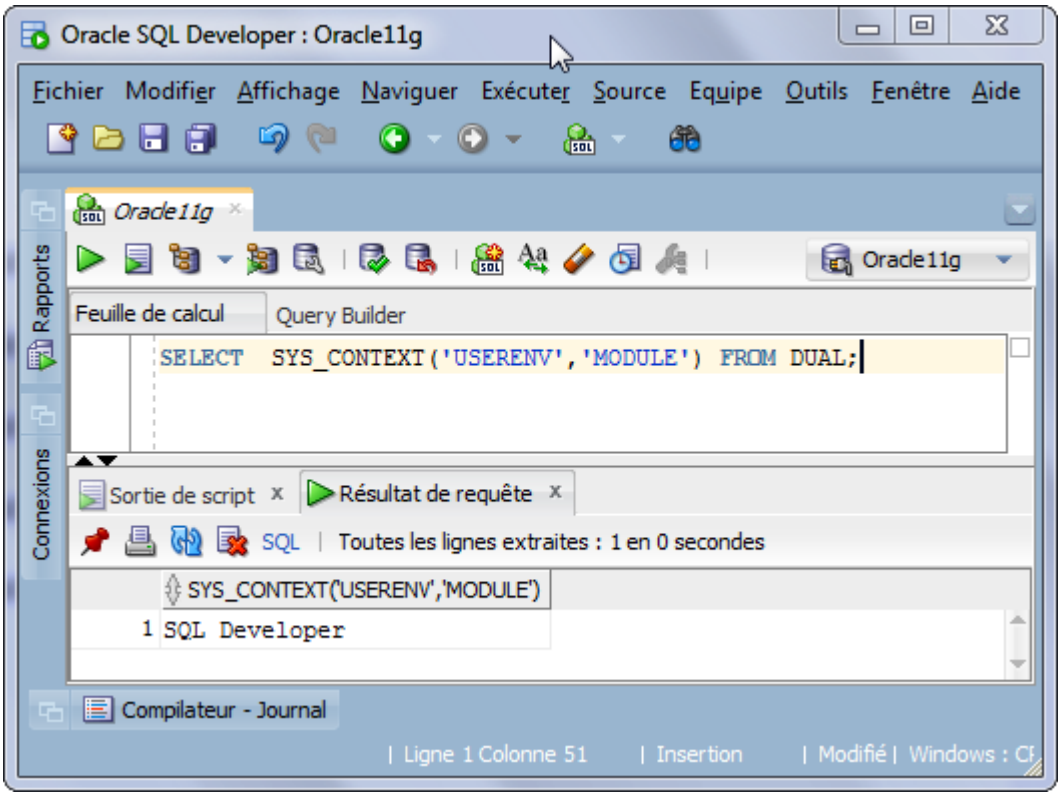


1.3 Les variables d'environnements

Il est possible de lire les variables d'environnements suivantes :

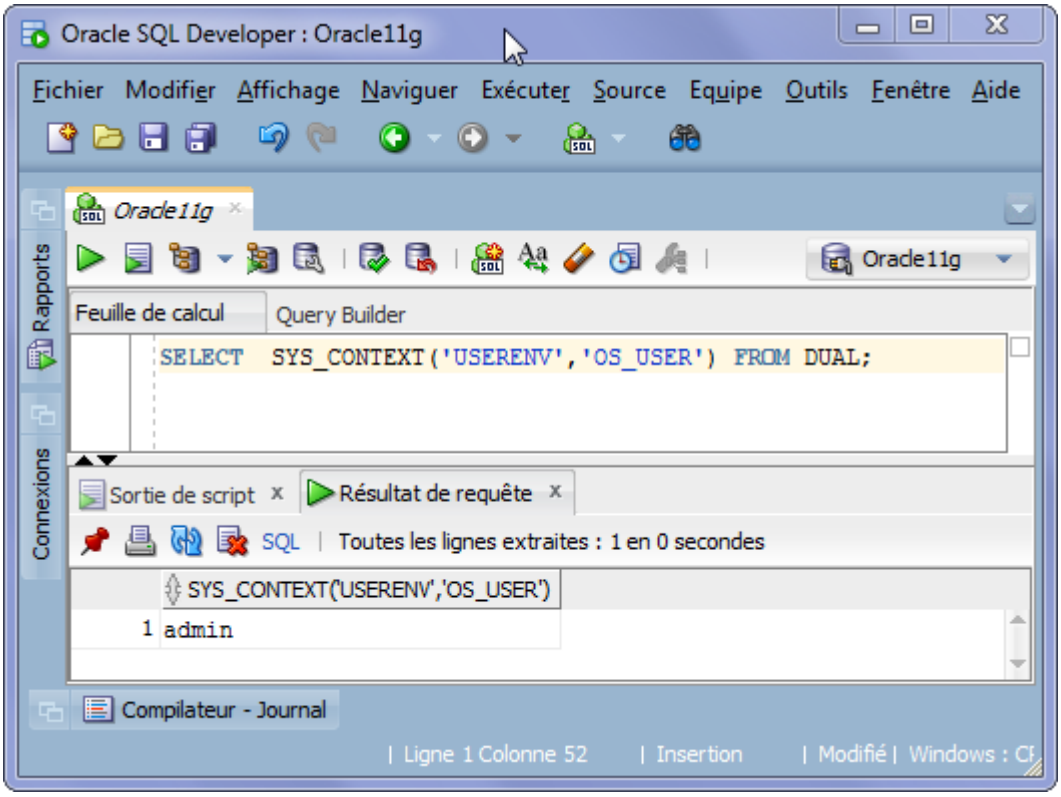
1) Variable MODULE

```
SELECT  SYS_CONTEXT('USERENV','MODULE') FROM DUAL;
```



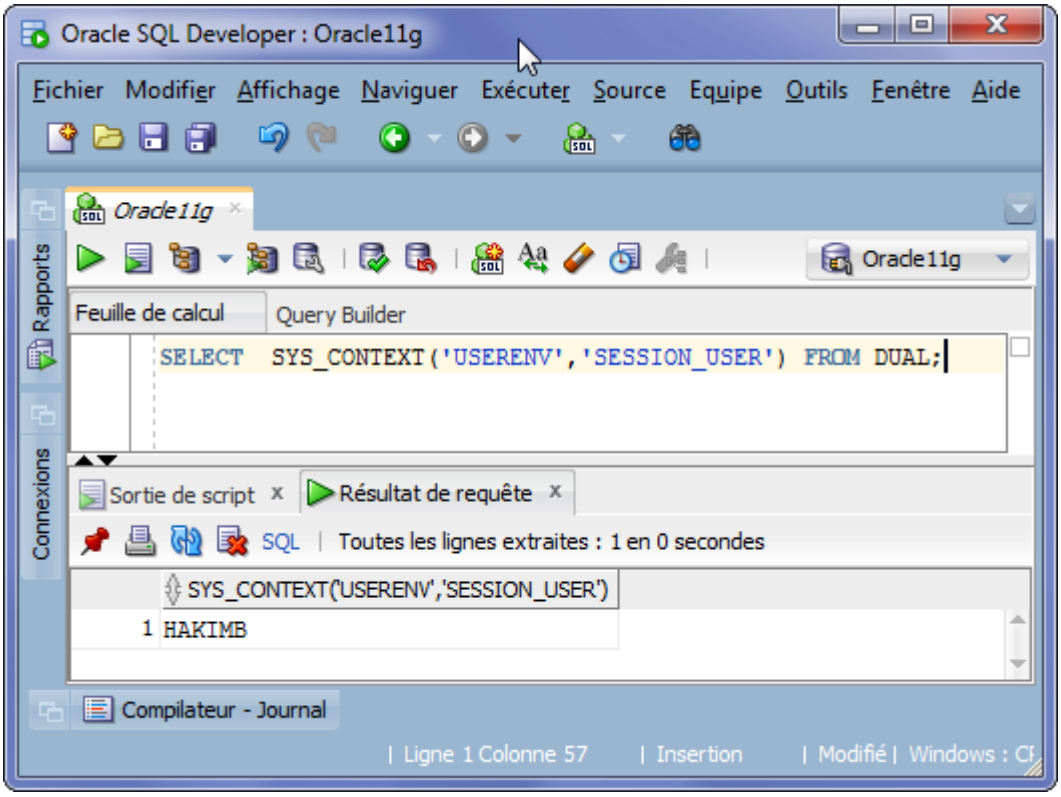
2) Variable OS_USER

```
SELECT  SYS_CONTEXT('USERENV','OS_USER') FROM DUAL;
```



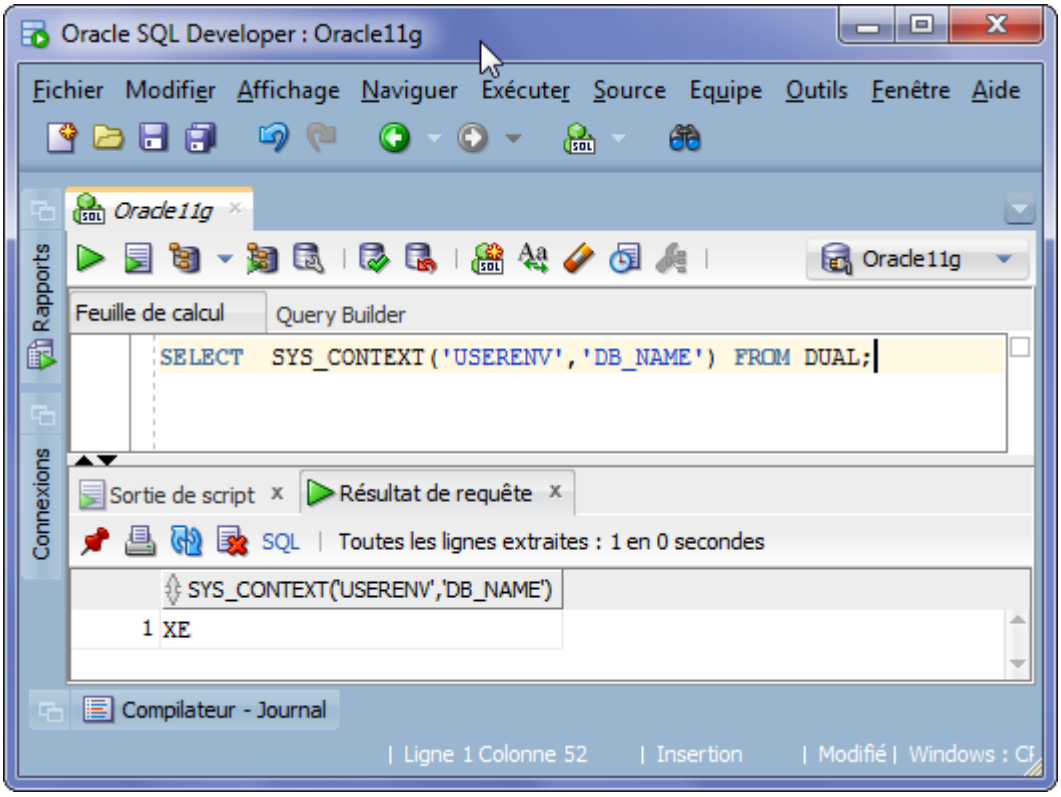
3) Variable `SESSION_USER`

```
SELECT  SYS_CONTEXT('USERENV','SESSION_USER') FROM DUAL;
```



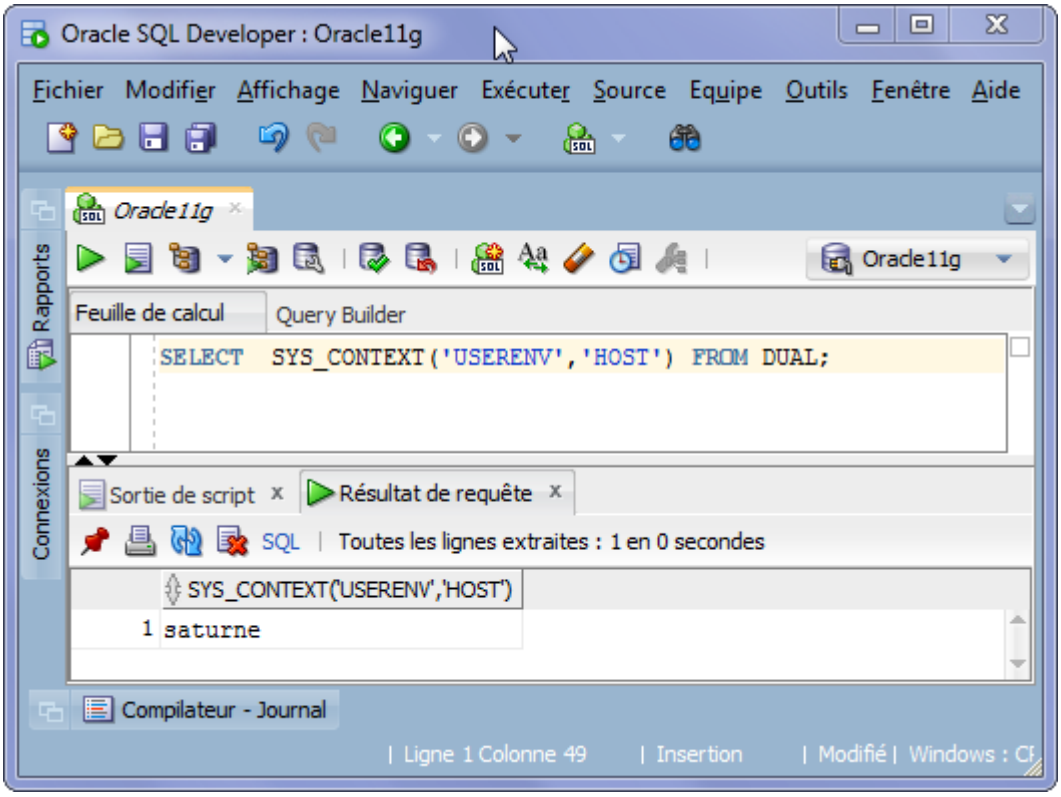
4) Variable DB_NAME

```
SELECT  SYS_CONTEXT('USERENV','DB_NAME') FROM DUAL;
```



5) Variable HOST

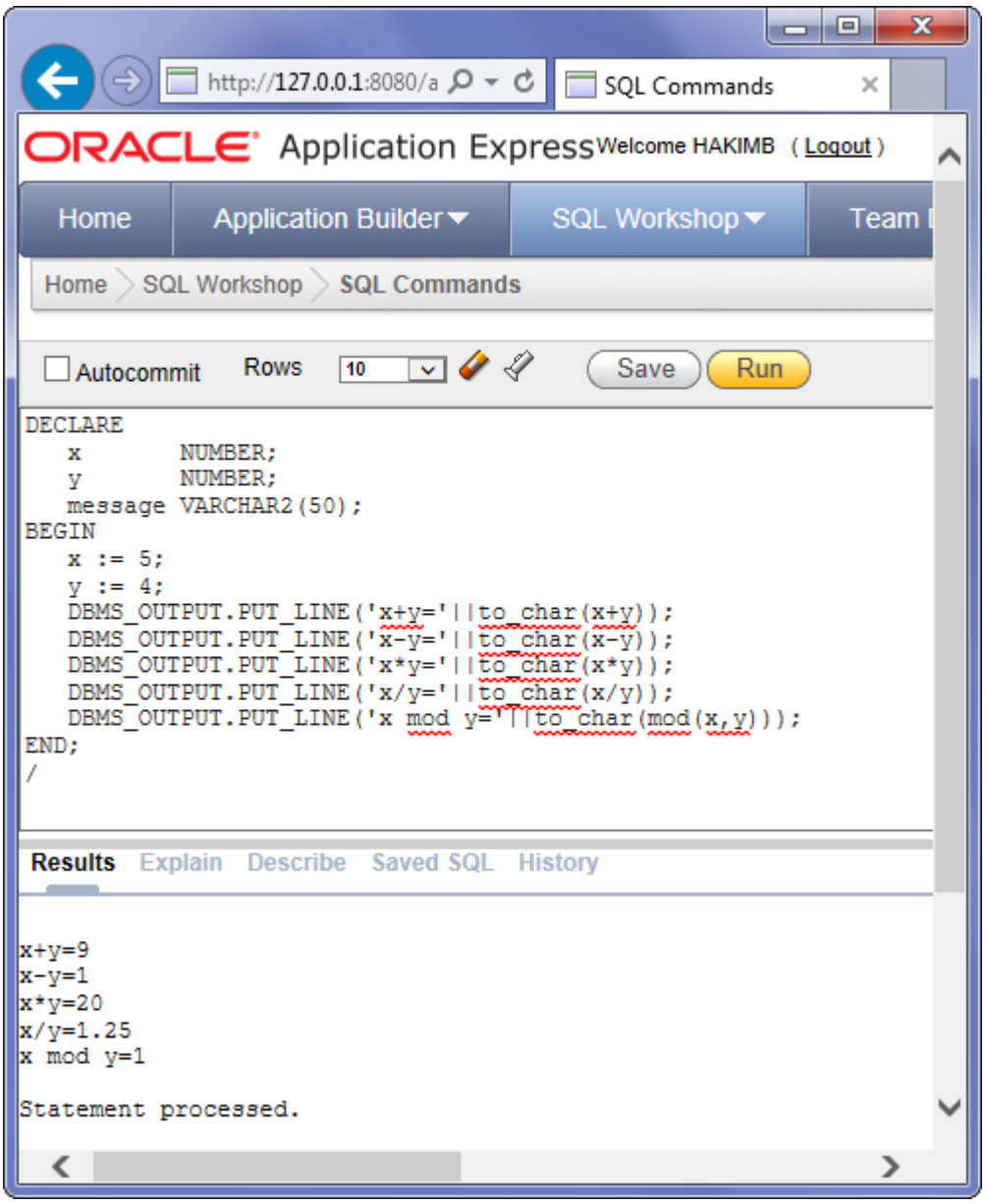
```
SELECT  SYS_CONTEXT('USERENV','HOST') FROM DUAL;
```



1.4 Les opérateurs

Permettent de créer des expressions complexes à partir d'expressions simples. Parmi les opérateurs, nous avons :

- opérateurs arithmétiques : **+** , **-** , **/** , ***** , **modulo**



Exécution du bloc PL/SQL sur la ligne de commande:

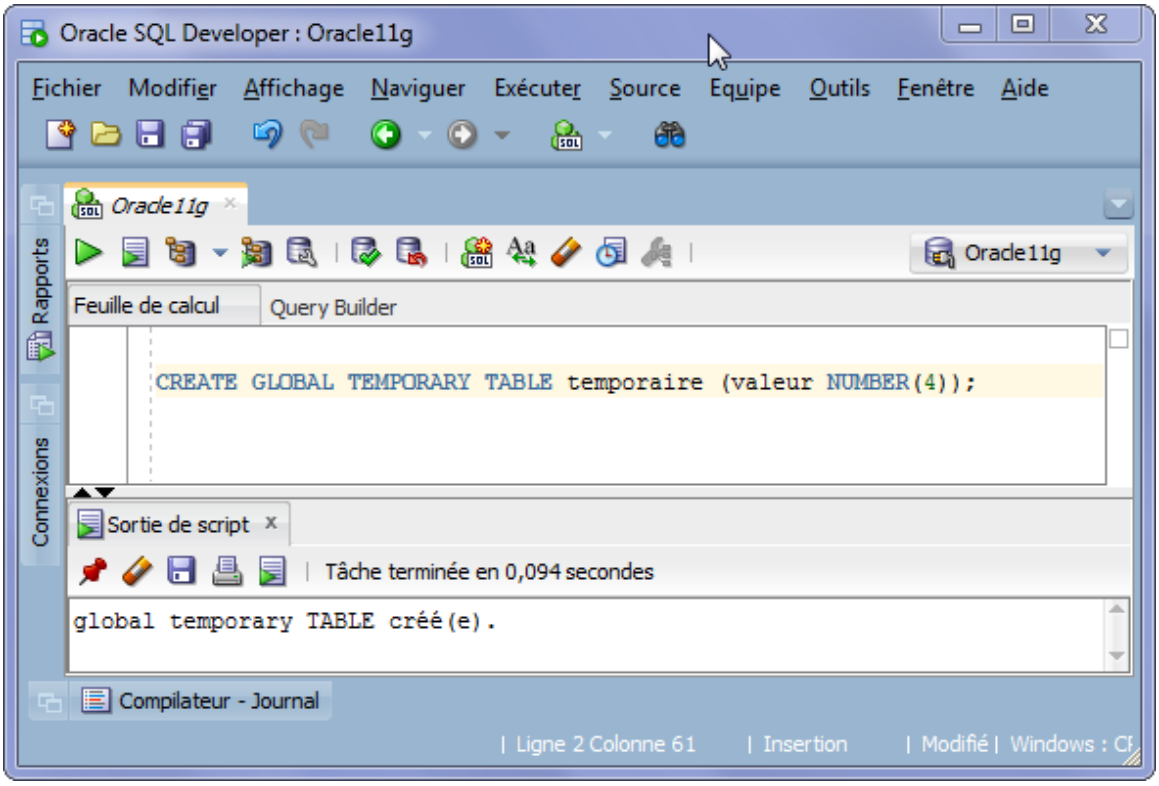
```
SQL> @test
x+y=9
x-y=1
x*y=20
x/y=1.25
x mod y=1

PL/SQL procedure successfully completed.
```

1.5 Les tables temporaires

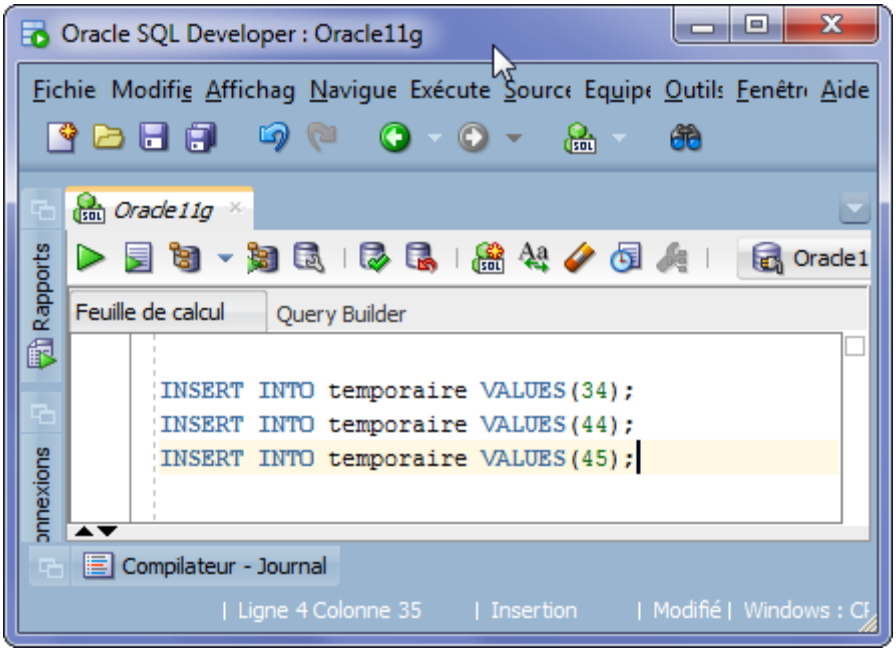
Une table temporaire est créée avec la commande :

```
CREATE GLOBAL TEMPORARY TABLE nom_table_temporaire
(
<Définitions des colonnes de la table>
)
[ON COMMIT DELETE ROWS;|ON COMMIT PRESERVE ROWS;]
```

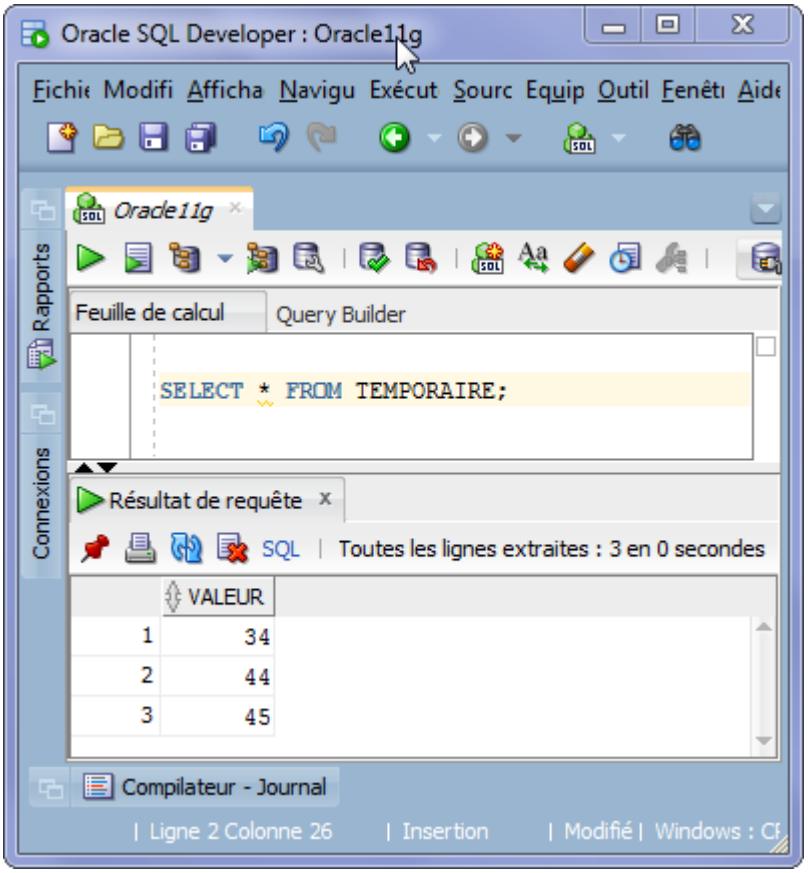
Par défaut c'est l'option **`ON COMMIT DELETE ROWS;`** qui est utilisé. Ce qui signifie que lorsque la transaction courante est complétée alors les données sont supprimées de la table.



Insérer des données dans la table temporaire :



Consulter la table temporaire :



Pratique1

Essayer d'accéder aux données de la table temporaire à partir d'une autre session.

Pratique2

Valider les trois INSERT (COMMIT) et vérifier si les données sont dans la table.

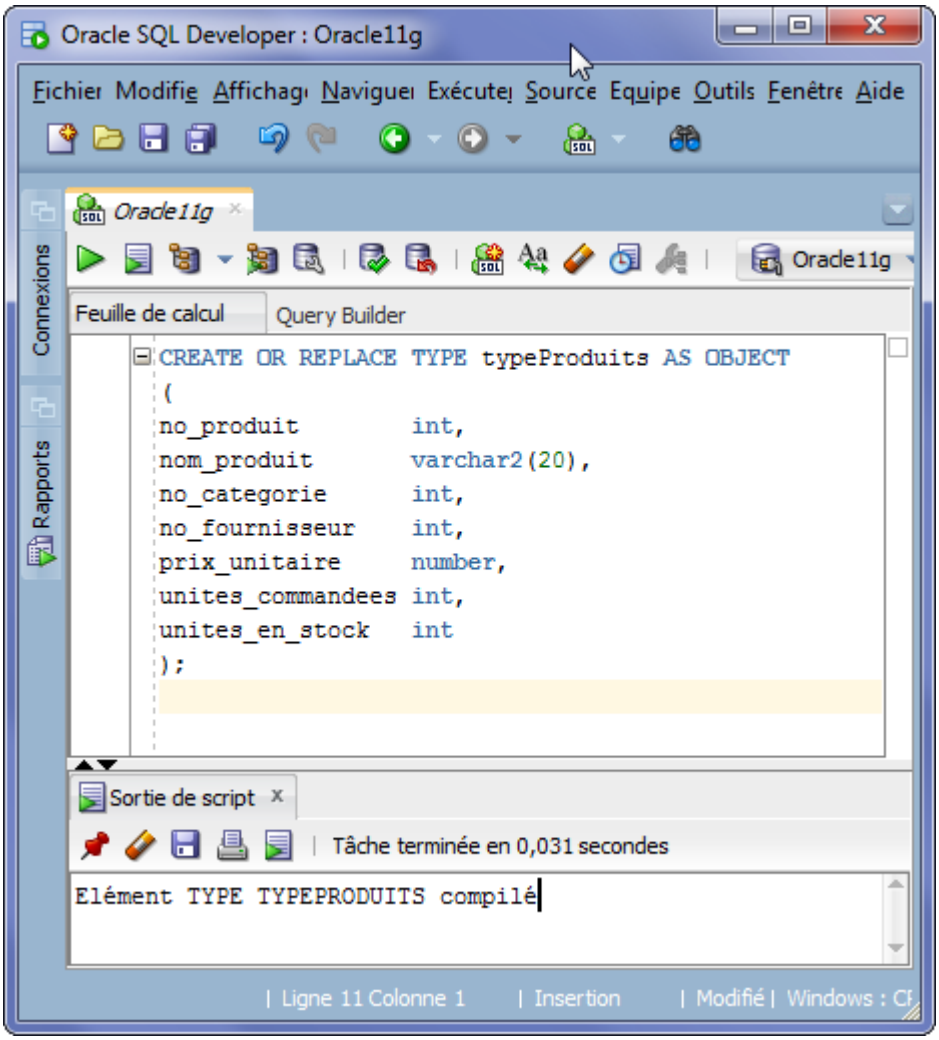
Pratique3

Fermer la session et ouvrir une nouvelle. Vérifier si la table temporaire existe. Si elle existe, vérifier si les données sont toujours dans la table.

1.6 Type défini par l'utilisateur

Il est possible de définir un nouveau type de données complexe avec la commande **CREATE TYPE** :

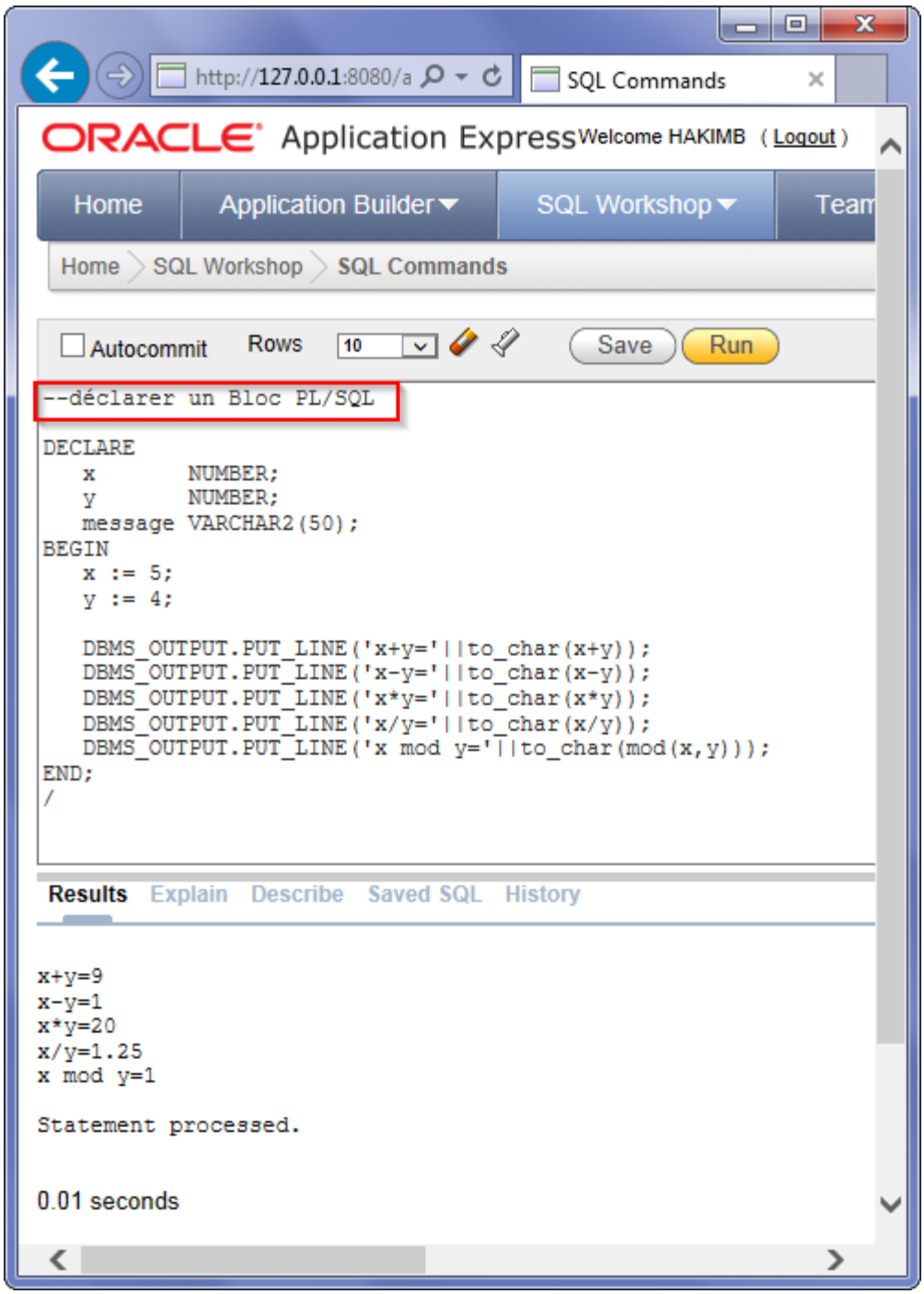
```
CREATE OR REPLACE TYPE typeProduits AS OBJECT
(
no_produit           int,
nom_produit          varchar2(20),
no_categorie         int,
no_fournisseur       int,
prix_unitaire        number,
unites_commandees    int,
unites_en_stock      int
);
```



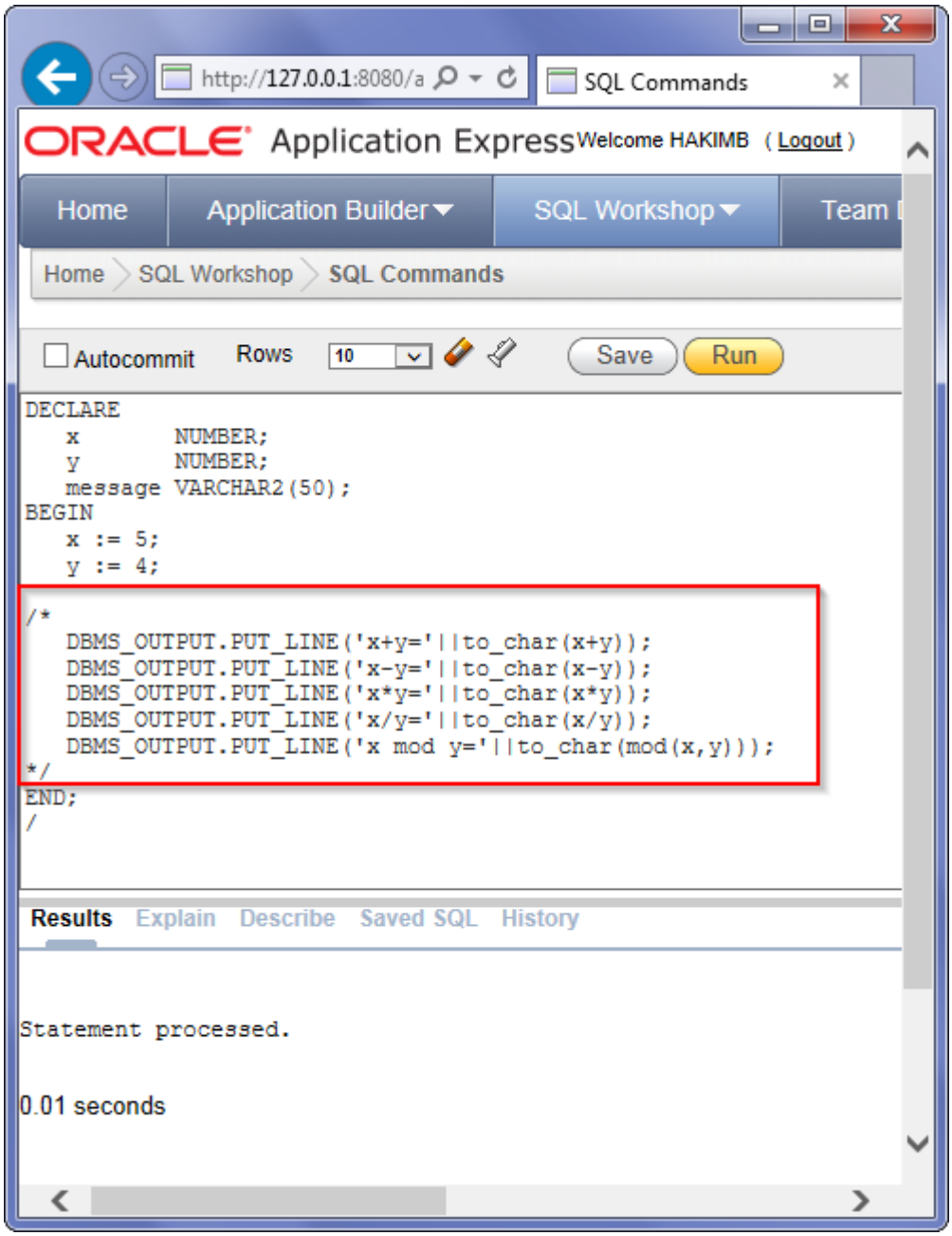
1.7 Les commentaires

Ce sont des parties de texte insérées dans un jeu d'instruction pour en expliquer le but. Les commentaires ne sont pas exécutés.

- **Commenter une ligne**



- **Commenter un bloc**



1.8 Les curseurs

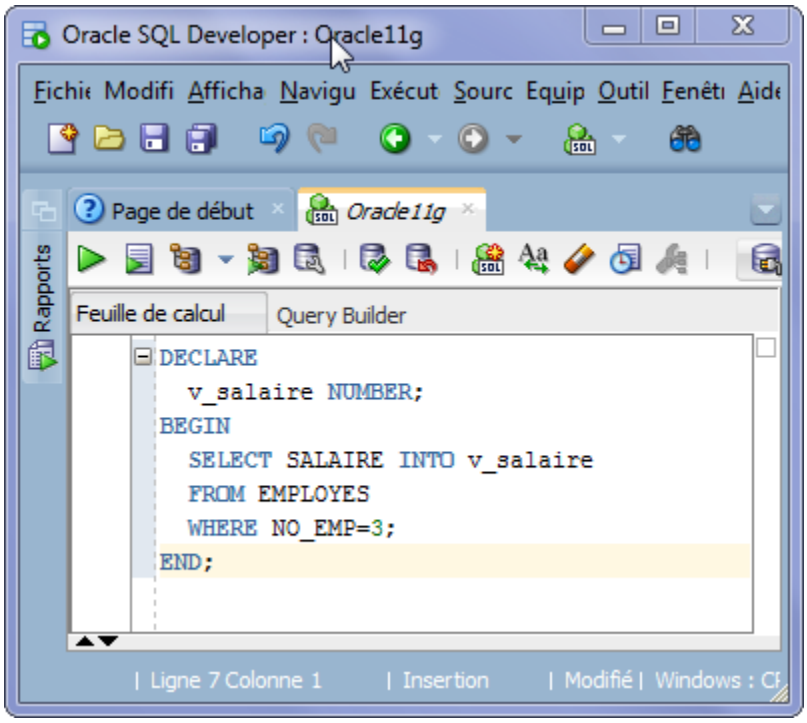
En PL/SQL on utilise l'instruction SELECT pour retrouver les données.

Ils existent deux facons pour récuperer les données d'un SELECT :

CURSEUR IMPLICITE

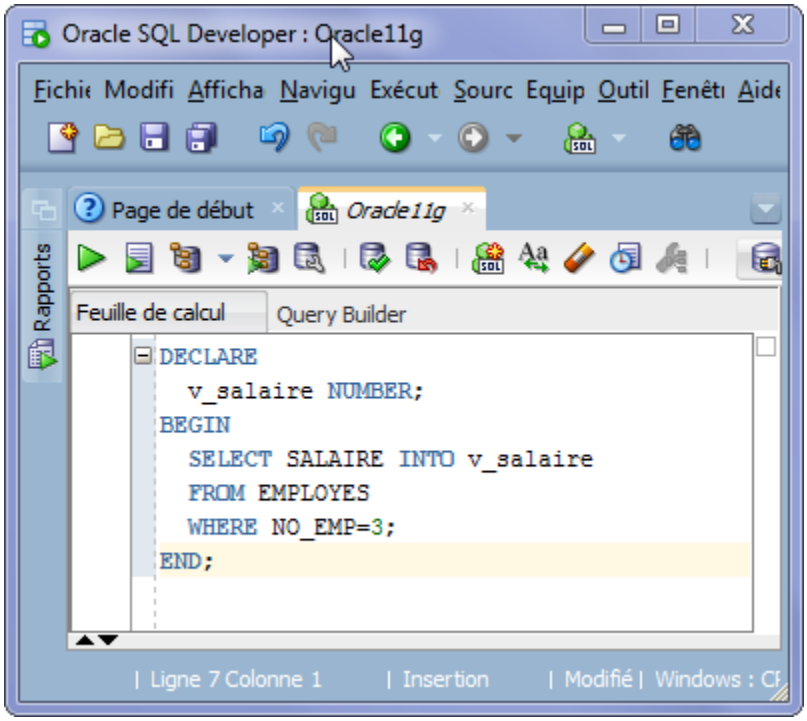
En utilisant l'option INTO de la commande SELECT, un curseur implicite est créé.

```
DECLARE
  v_salaire NUMBER;
BEGIN
  SELECT SALAIRE INTO v_salaire
  FROM EMPLOYES
  WHERE NO_EMP=3;
END;
```



CURSEUR EXPLICITE

Un curseur explicite doit etre declarer avant d'etre utilisé.

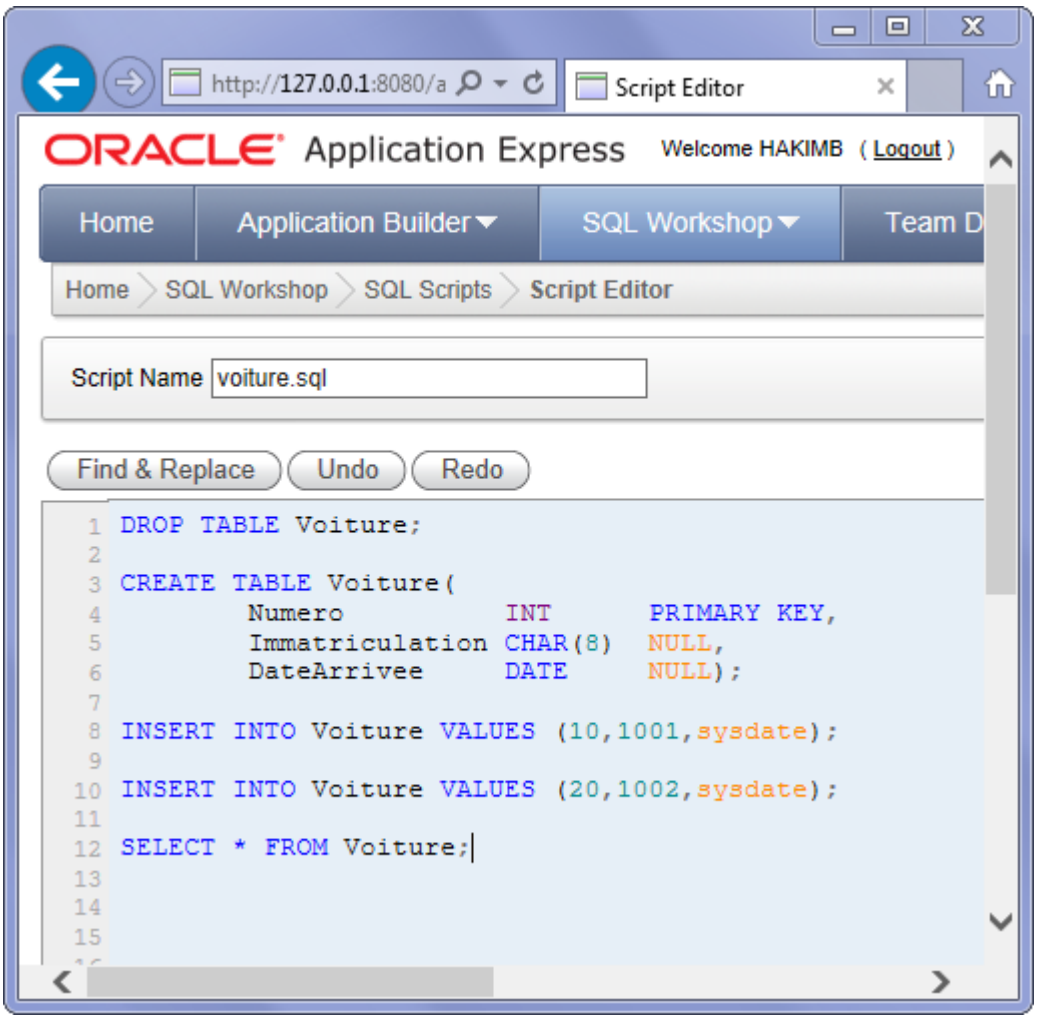



```
DECLARE
  prix  PRODUITS.PRIX_UNITAIRE%TYPE;
  CURSOR curseur_produits IS
  SELECT PRIX_UNITAIRE
  FROM PRODUITS;
BEGIN
  OPEN curseur_produits;
  FETCH curseur_produits INTO prix;
  CLOSE curseur_produits;
END;
```

1.9 Les scripts

C'est un ensemble d'instructions PL/SQL :

Script **voiture.sql :**



Exécution du script sur la ligne de commande :

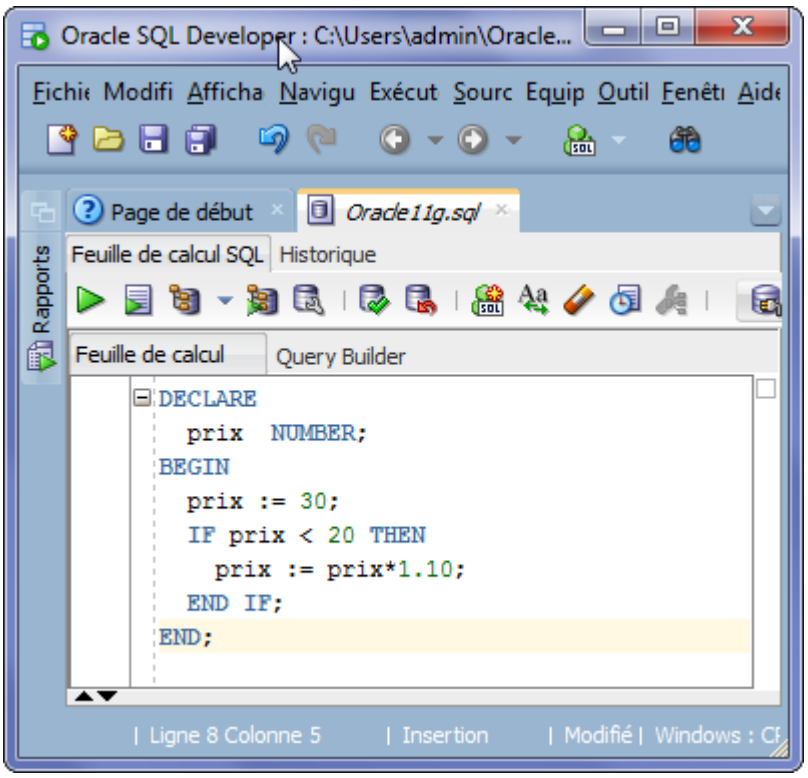
```
SQL> @voiture.sql

Table dropped.

Table created.

1 row created.

1 row created.

    NUMERO IMMATRIC DATEARRIVEE
---------- -------- ---------------
        10 1001     04-jan-13
        20 1002     04-jan-13
```

2 INSTRUCTIONS CONDITIONELLES

2.1 IF..THEN

Syntaxe :

```
IF <condition> THEN
   <Code PL/SQL si condition est vraie>
END IF;
```

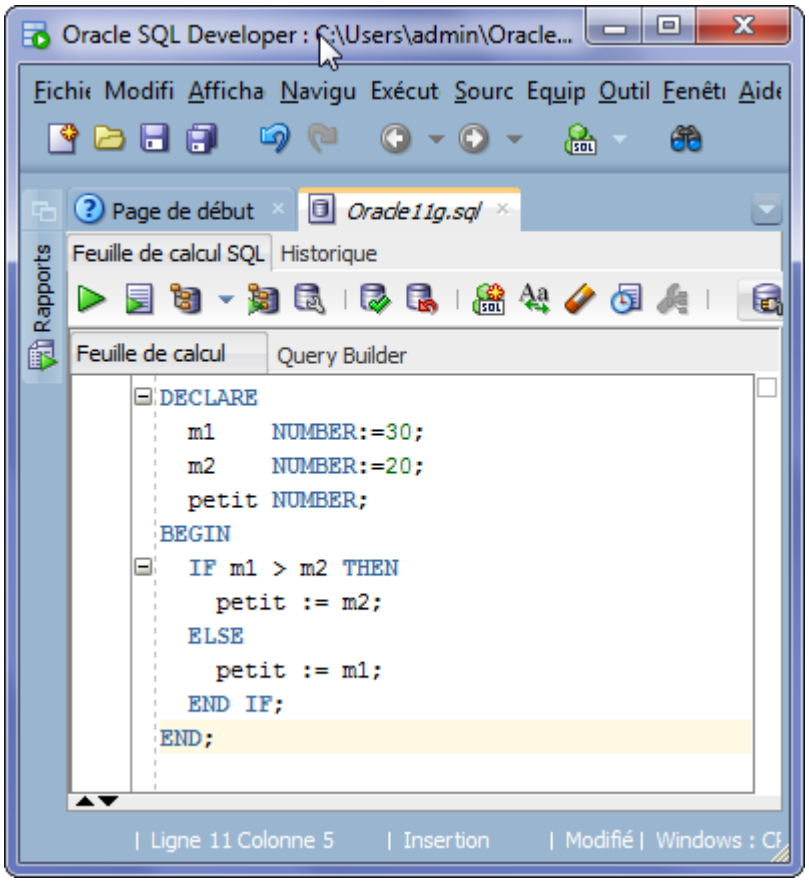


2.2 INSTRUCTION IF

2.2.1 IF..THEN..ELSE

Syntaxe :

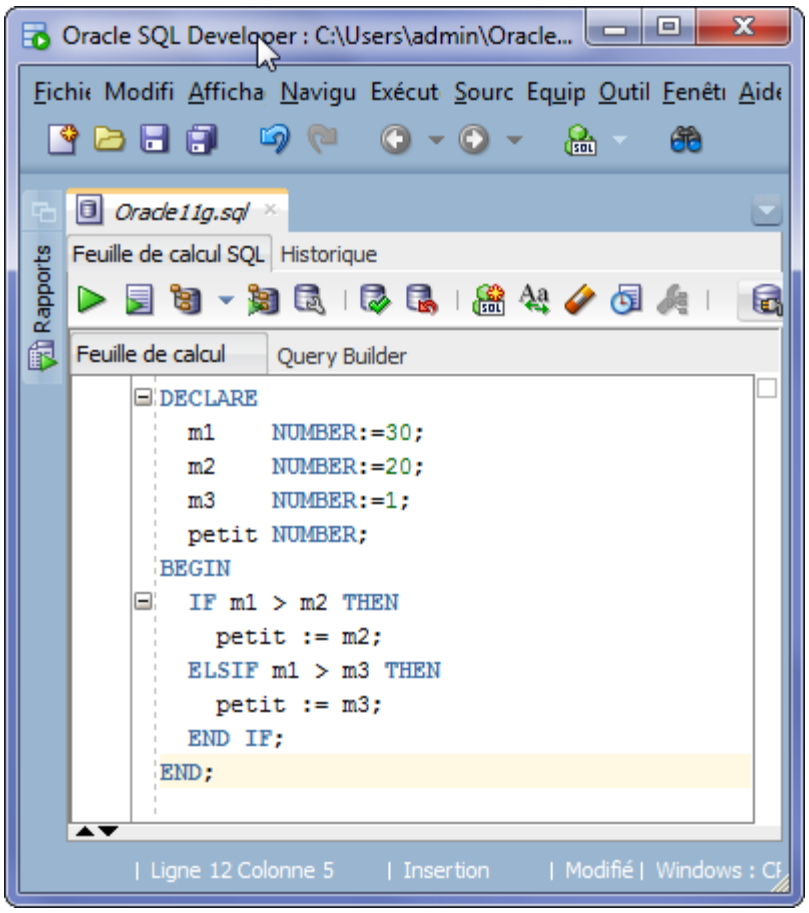
```
IF <condition> THEN
   <Code PL/SQL si condition est vraie>
ELSE
   <Code PL/SQL si condition est fausse>
END IF;
```



2.2.2 IF..THEN..ELSEIF

Syntaxe :

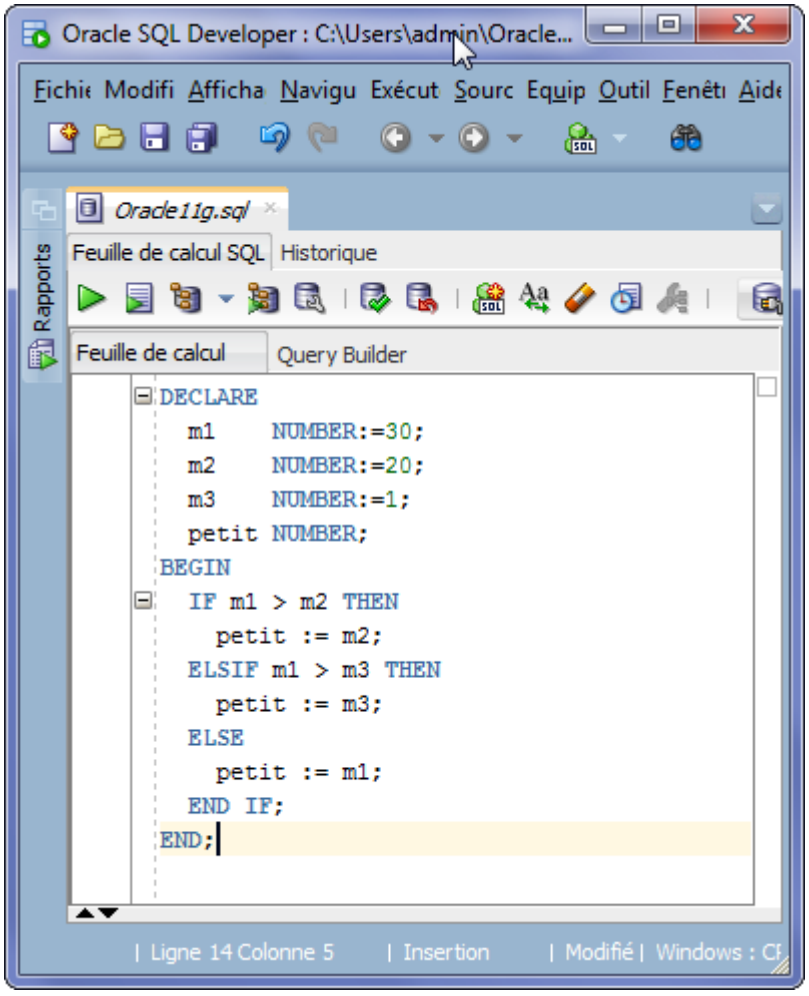
```
IF <condition1> THEN
   <Code PL/SQL si condition1 est vraie>
ELSIF <condition2> THEN
   <Code PL/SQL si condition1 est fausse
                et condition2 est vraie >
END IF;
```



2.2.3 IF..THEN..ELSEIF..ELSE

Syntaxe :

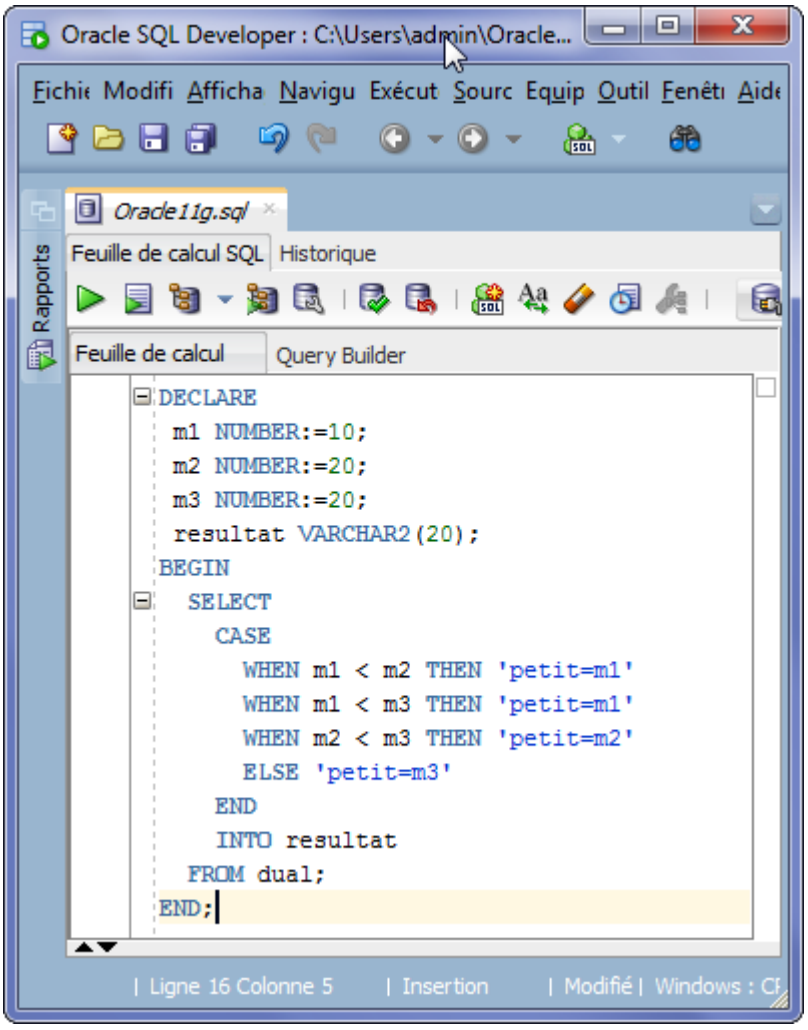
```
IF <condition1> THEN
   <Code PL/SQL si condition1 est vraie>
ELSIF <condition2> THEN
   <Code PL/SQL si condition1 est fausse
                et condition2 est vraie >
ELSE
<Code PL/SQL si condition1 et condition2 sont fausses>

END IF;
```



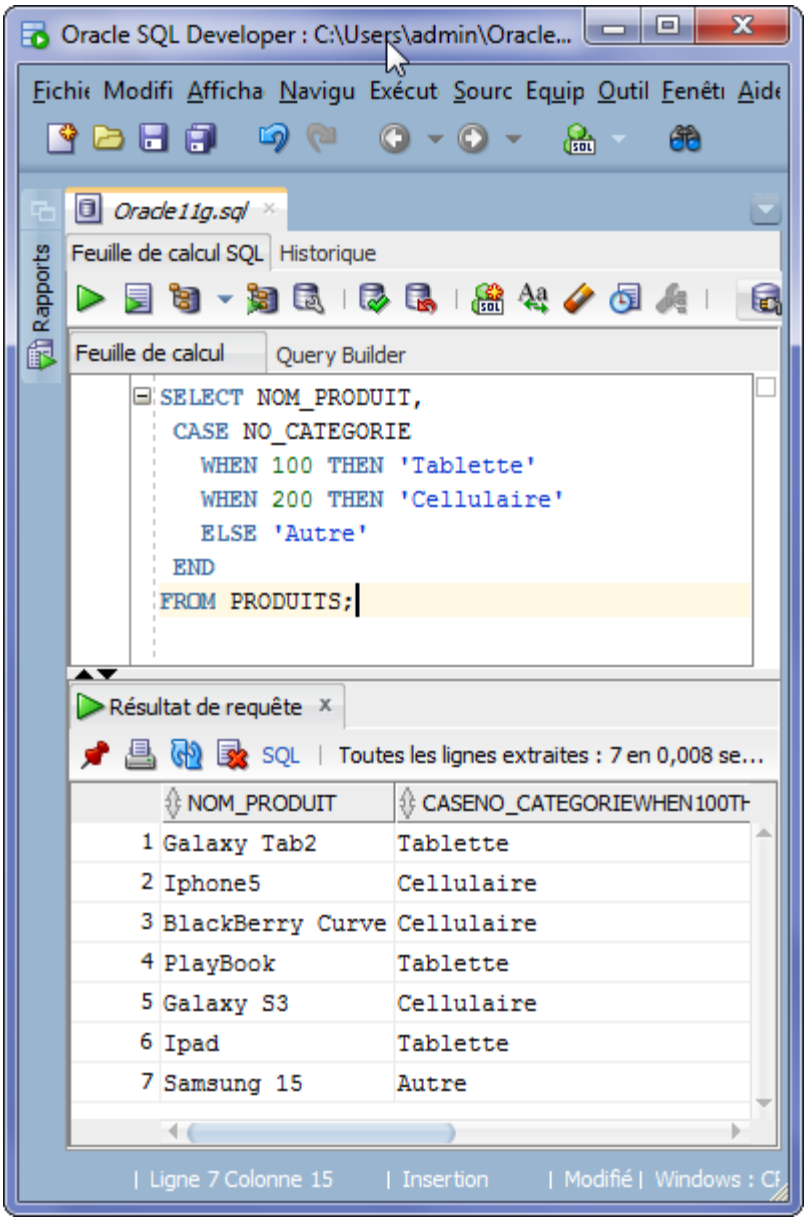
2.3 INSTRUCTION CASE

Syntaxe :

```
CASE [ expression ]
   WHEN condition1 THEN résultat1
   WHEN condition2 THEN résultat2
   ...
   WHEN conditionn THEN résultatn

   ELSE résultat
END
```



```
SELECT NOM_PRODUIT,
 CASE NO_CATEGORIE
   WHEN 100 THEN 'Tablette'
   WHEN 200 THEN 'Cellulaire'
   ELSE 'Autre'
 END
FROM PRODUITS;
```



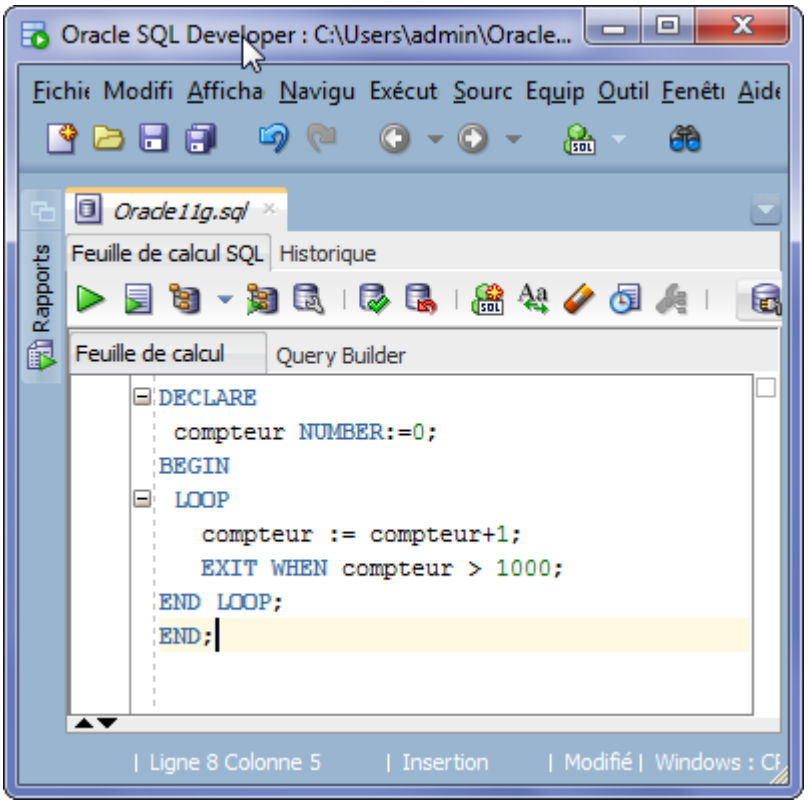
3 INSTRUCTIONS RÉPÉTITIVES

3.1 INSTRUCTION LOOP

Syntaxe :

```
LOOP
   <Code PL/SQL>
   EXIT [ WHEN condition ];
END LOOP;
```

On utilise la commande EXIT pour sortir de la boucle.



3.2 INSTRUCTION FOR

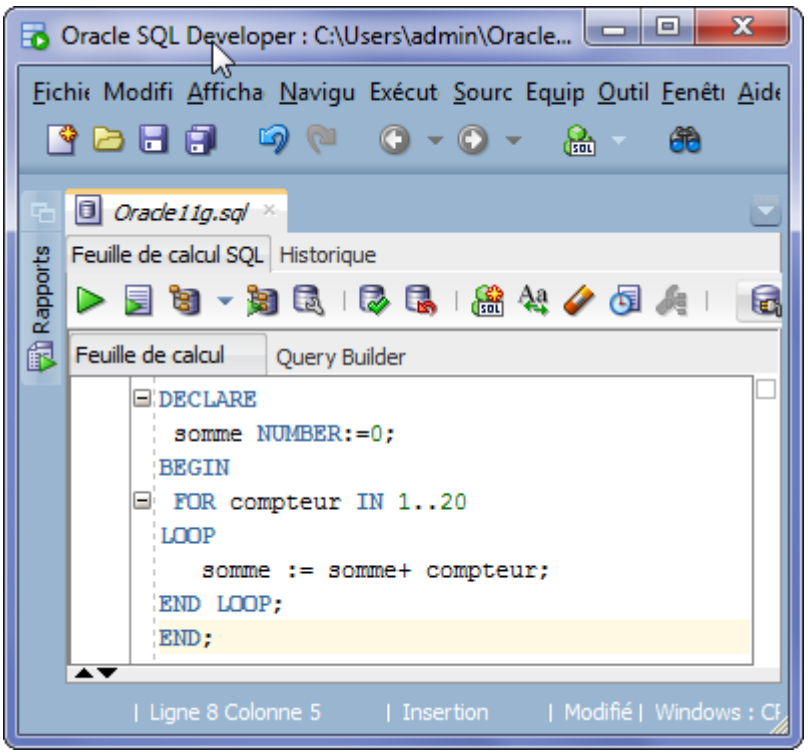
Syntaxe :

```
FOR compteur_boucle IN [REVERSE]
valeur_debut..valeur_fin
LOOP
   <Code PL/SQL>
END LOOP;
```

Le compteur de la boucle n'a pas besoin d'être déclarée.

Dans ce qui suit, on ne déclare pas la variable compteur.

```
DECLARE
 somme NUMBER:=0;
BEGIN
 FOR compteur IN 1..20
 LOOP
    somme := somme+ compteur;
 END LOOP;
END;
```



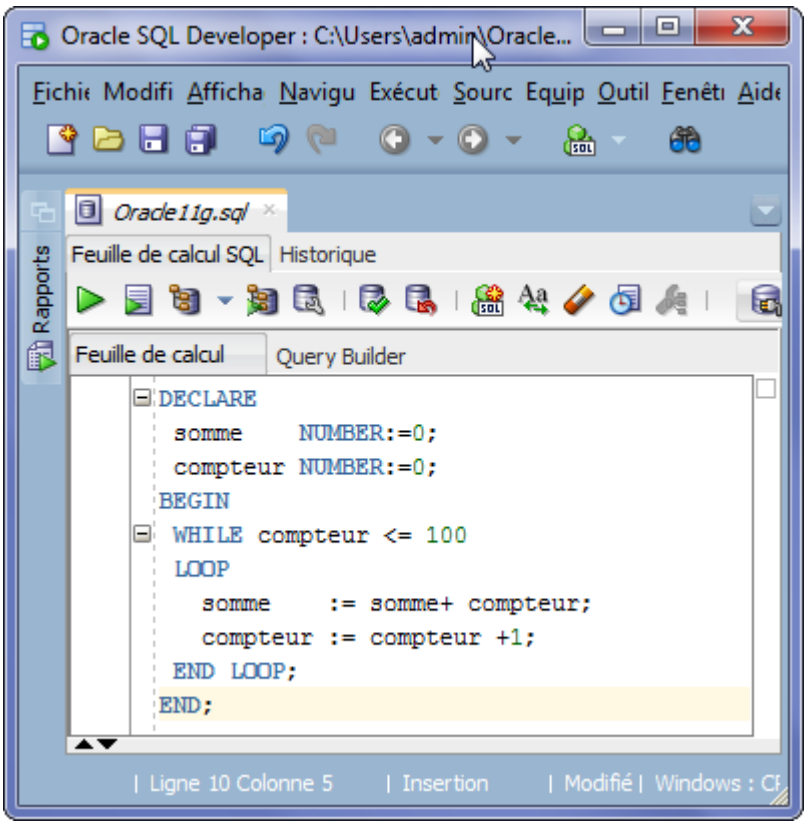
3.3 INSTRUCTION WHILE

Syntaxe :

```
WHILE condition
LOOP
   <Code PL/SQL>
END LOOP;
```

EXEMPLE

```
DECLARE
 somme     NUMBER:=0;
 compteur NUMBER:=0;
BEGIN
 WHILE compteur <= 100
 LOOP
   somme     := somme+ compteur;
   compteur := compteur +1;
 END LOOP;
END;
```



4 LES PROCÉDURES STOCKÉES

Une procédure stockée est un ensemble nommé d'instructions **PL/SQL**, précompilée et stockée sur le serveur. Les procédures stockées constituent une méthode privilégiée pour encapsuler les tâches répétitives afin de les exécuter efficacement.

Elles prennent en charge les variables déclarées par l'utilisateur, le contrôle de flux et d'autres fonctionnalités de programmation avancées.

Ce sont de véritables programmes pouvant :

- recevoir des paramètres
- renvoyer des valeurs
- être exécutés à distance
- ayant leurs propres droits d'accès (privilège EXECUTE).

De plus, les procédures stockées sont stockées dans le cache mémoire de la base de données sous forme compilée lors de leur première exécution, ce qui accroît les performances pour les exécutions suivantes.

Une procédure stockée peut être définie comme une suite d'instructions PL/SQL, stockée dans la base de données et identifié par son nom.

Pour permettre à cette suite d'instructions de s'adapter au plus grand nombre de cas, certaines valeurs du code sont paramétrables lors de l'appel de la procédure.

Avantage des procédures stockées

Les procédures stockées offrent de nombreux avantages :

1) **Performance**

   Chaque fois qu'une requête **PL/SQL** est exécutée, le serveur détermine si la syntaxe est correcte puis il construit un plan d'exécution avant d'exécuter la requête.

   Les procédures stockées sont plus performantes parce que le serveur vérifie la syntaxe à la création. La première fois que la procédure stockée est exécutée, le serveur créé le plan d'exécution et compile la procédure. Les exécutions ultérieures de cette procédure stockée seront plus rapides parce que le serveur ne fera pas de nouvelles vérifications sur la syntaxe et la construction du plan d'exécution ;

2) **Réutilisabilité**

   Une fois que la procédure stockée est créée, vous pouvez l'appeler à n'importe quel moment. Ceci permet une modularité et encourage la réutilisation de votre code ;

3) **Simplification**

   Elles peuvent partager une logique d'application avec d'autres applications, contribuant ainsi à garantir la cohérence des accès aux données et de leurs modifications.

   Elles peuvent encapsuler une fonctionnalité d'entreprise. Les règles et politiques de fonctionnement encapsulées dans les procédures stockées peuvent être modifiées au même endroit.

   Tous les clients peuvent utiliser les mêmes procédures stockées afin de garantir la cohérence des accès aux données et de leurs modifications ;

4) Accès Réseau

   Elles contribuent à la réduction du trafic sur le réseau. Au lieu d'envoyer des centaines d'instructions **PL/SQL** sur le réseau, les utilisateurs peuvent effectuer une opération complexe à l'aide d'une seule instruction, réduisant ainsi le nombre de demandes échangées entre le client et le serveur.

4.1 Création d'une procédure

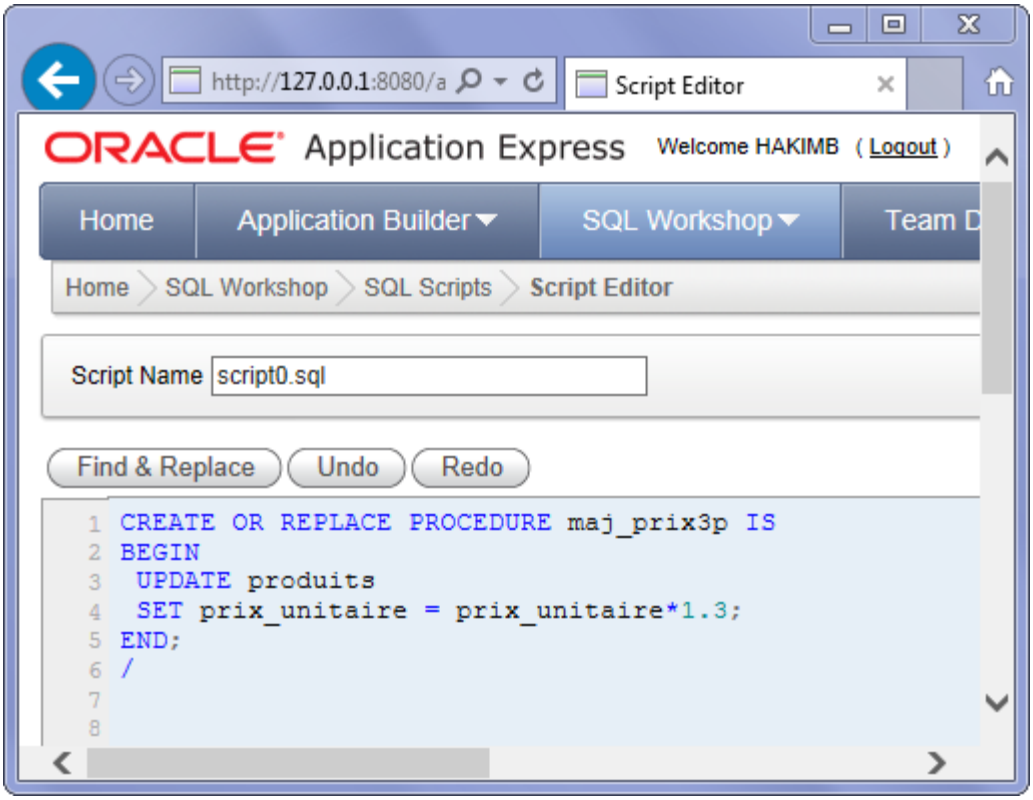
La création d'une procédure stockée se fait par l'instruction CREATE PROCEDURE.

La syntaxe de l'instruction **CREATE PROCEDURE** est la suivante :

```
CREATE [OR REPLACE] PROCEDURE nom_procédure
    [ (paramètre [,paramètre]) ]
IS
    [<Section déclaration des variables>]
BEGIN
    <CODE PL/SQL>
[EXCEPTION
    <Section gestion des exceptions>]
END [nom_procédure];
```

EXEMPLE

Écrire une procédure **maj_prix3p** qui permet d'augmenter le prix de chaque produit de 3% :



4.2 Appel d'une procédure

Les procédures stockées peuvent être exécutées par simple appel de leur nom ou par l'instruction EXECUTE.

EXEMPLE

Exécuter la procédure **maj_prix3p** :

```
SQL> EXECUTE maj_prix3p

PL/SQL procedure successfully completed.
```

```
SQL> ROLLBACK;

Rollback complete.
```

4.3 Utilisation des paramètres

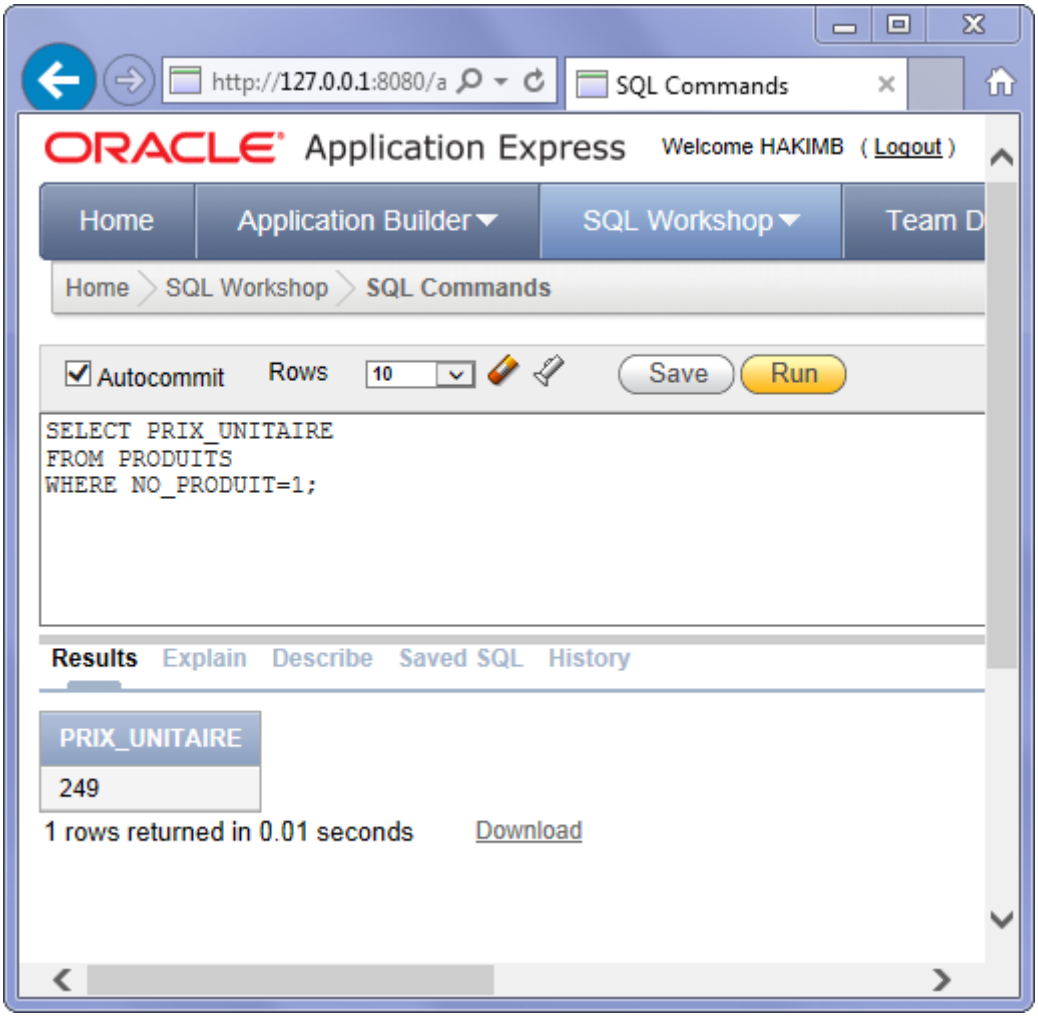
Pour permettre à une procédure de s'adapter au plus grand nombre de cas, certaines valeurs du code sont paramétrables lors de l'appel de la procédure. Les paramètres peuvent être en entré (INPUT) ou sortie (OUTPUT).

EXEMPLE1 (INPUT)

Écrire une procédure maj_prix qui permet de modifier le prix d'un produit donné.

On va utiliser deux paramètres INPUT pour passer les valeurs du nouveau prix et du numéro de produit :

Valeur du prix avant l'exécution de la procédure **maj_prix :**

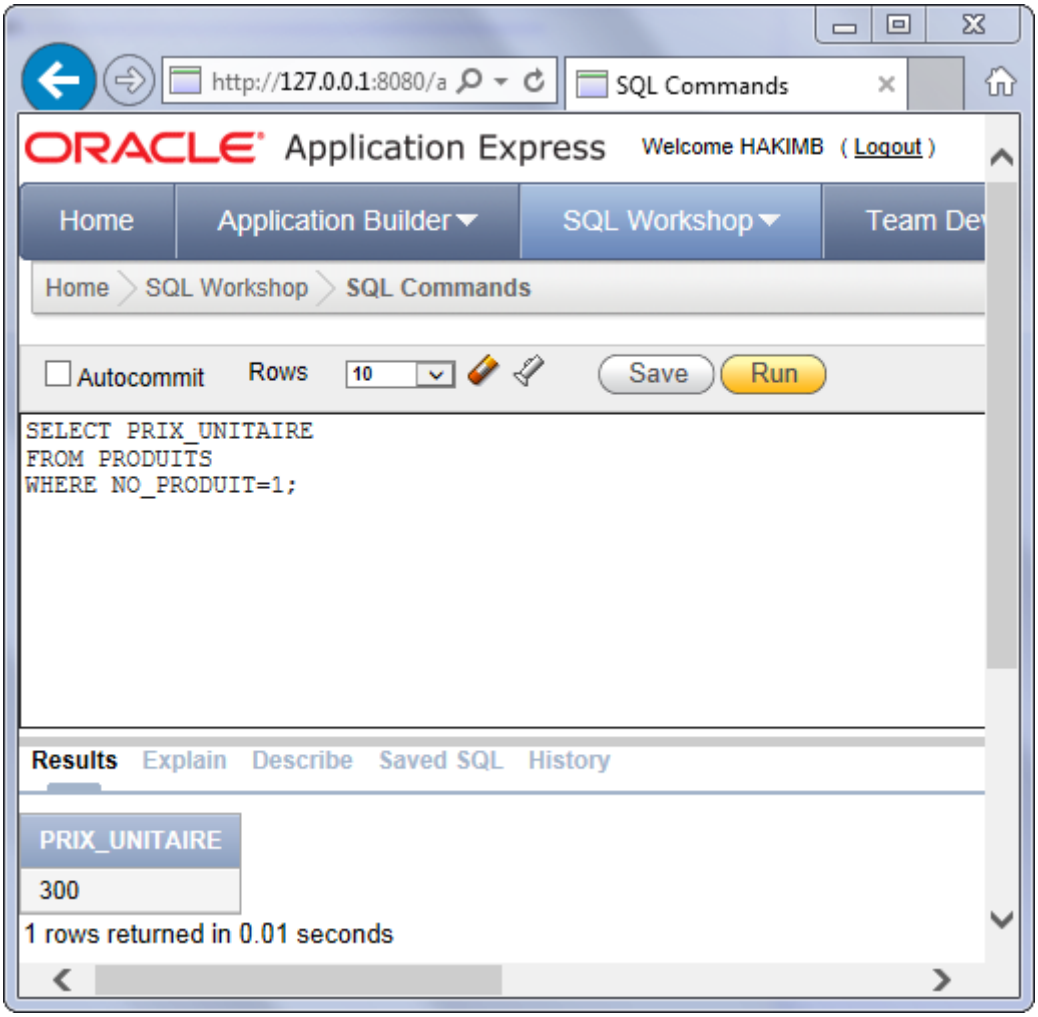


Exécuter la procédure **maj_prix** :

```
SQL> EXECUTE maj_prix(300,1)

PL/SQL procedure successfully completed.
```

```
SQL> COMMIT;

Commit complete.
```

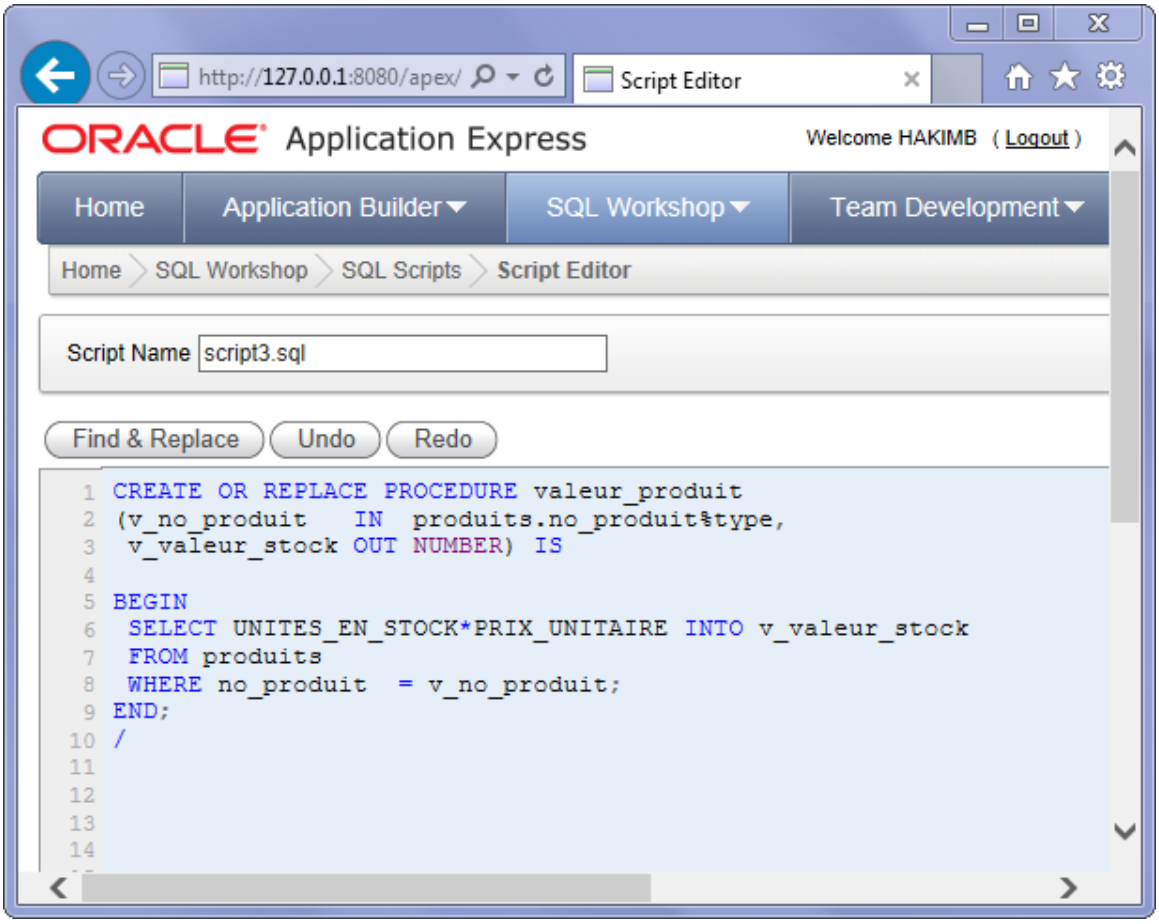
Valeur du prix après l'exécution de la procédure **maj_prix :**



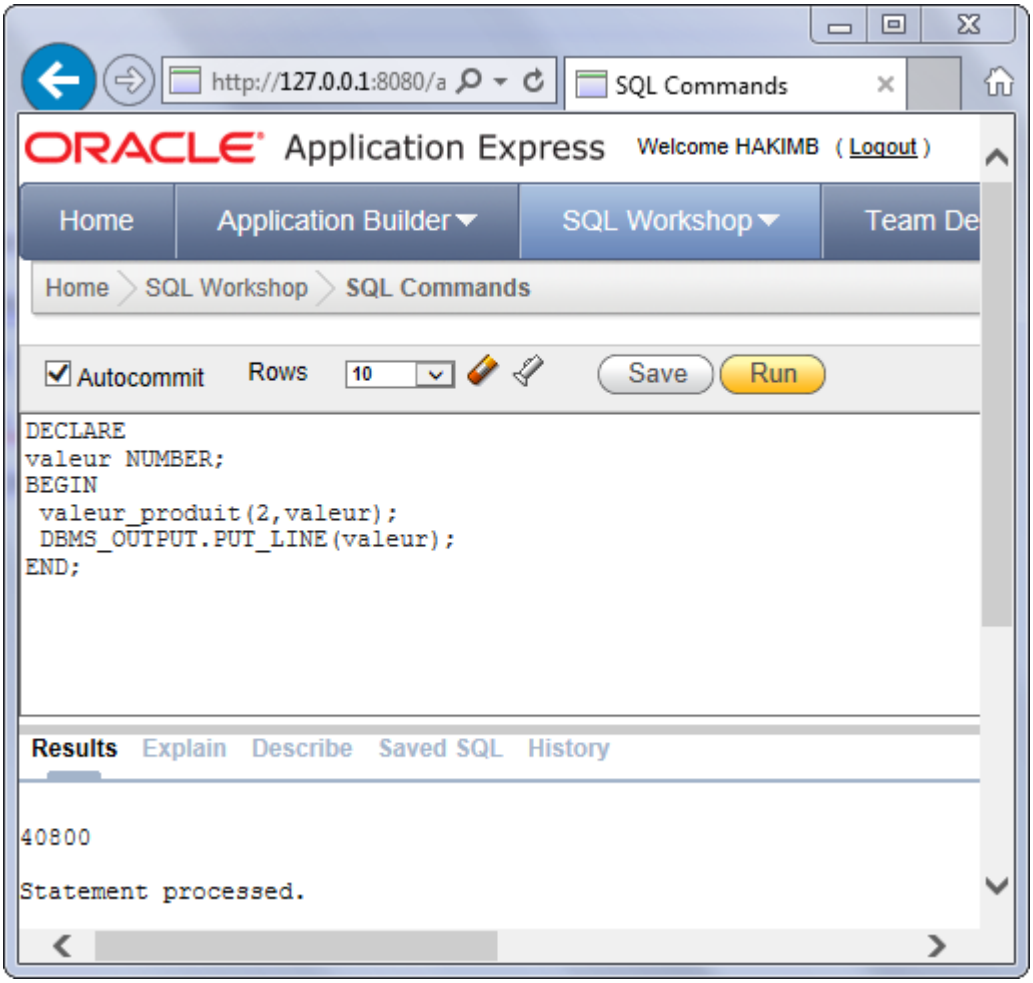
EXEMPLE2 (INPUT et OUTPUT)

Écrire une procédure **valeur_produit** calcul la valeur en stock d'un produit donné.

Le numéro de commande est fourni en paramètre IN à la procédure. La valeur du stock est un paramètre OUT.

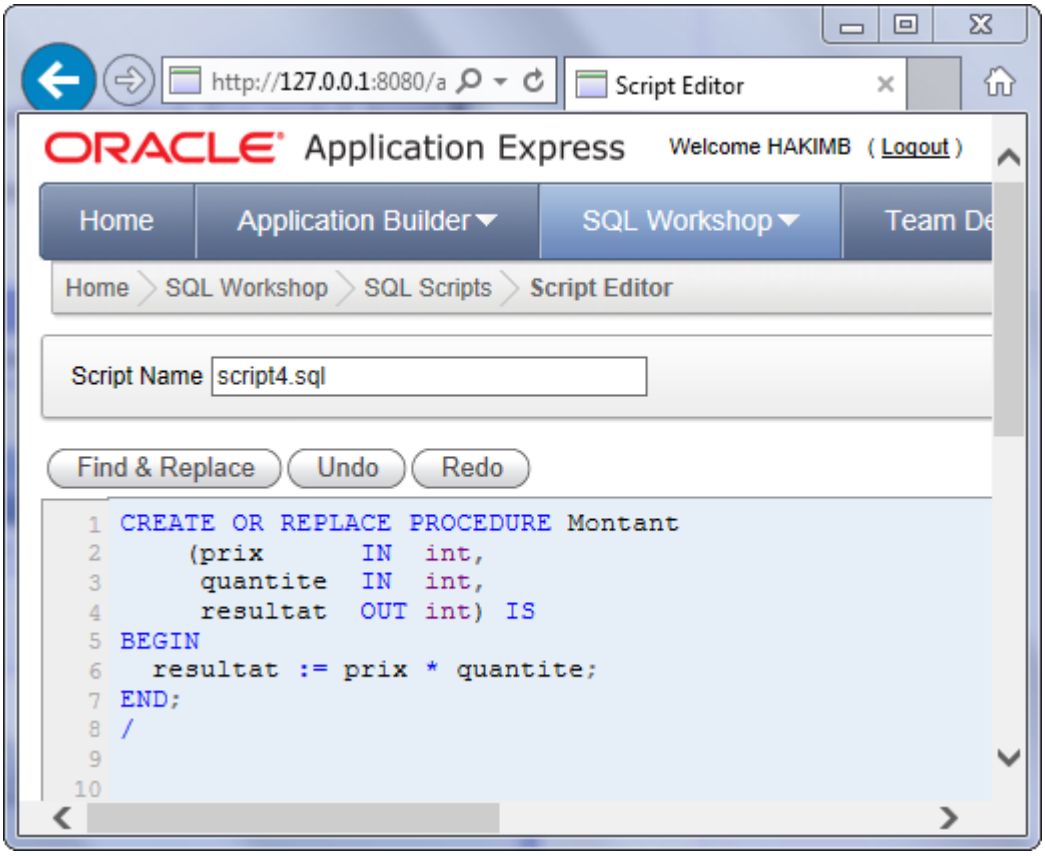


Exécution de la procédure **valeur_produit** :

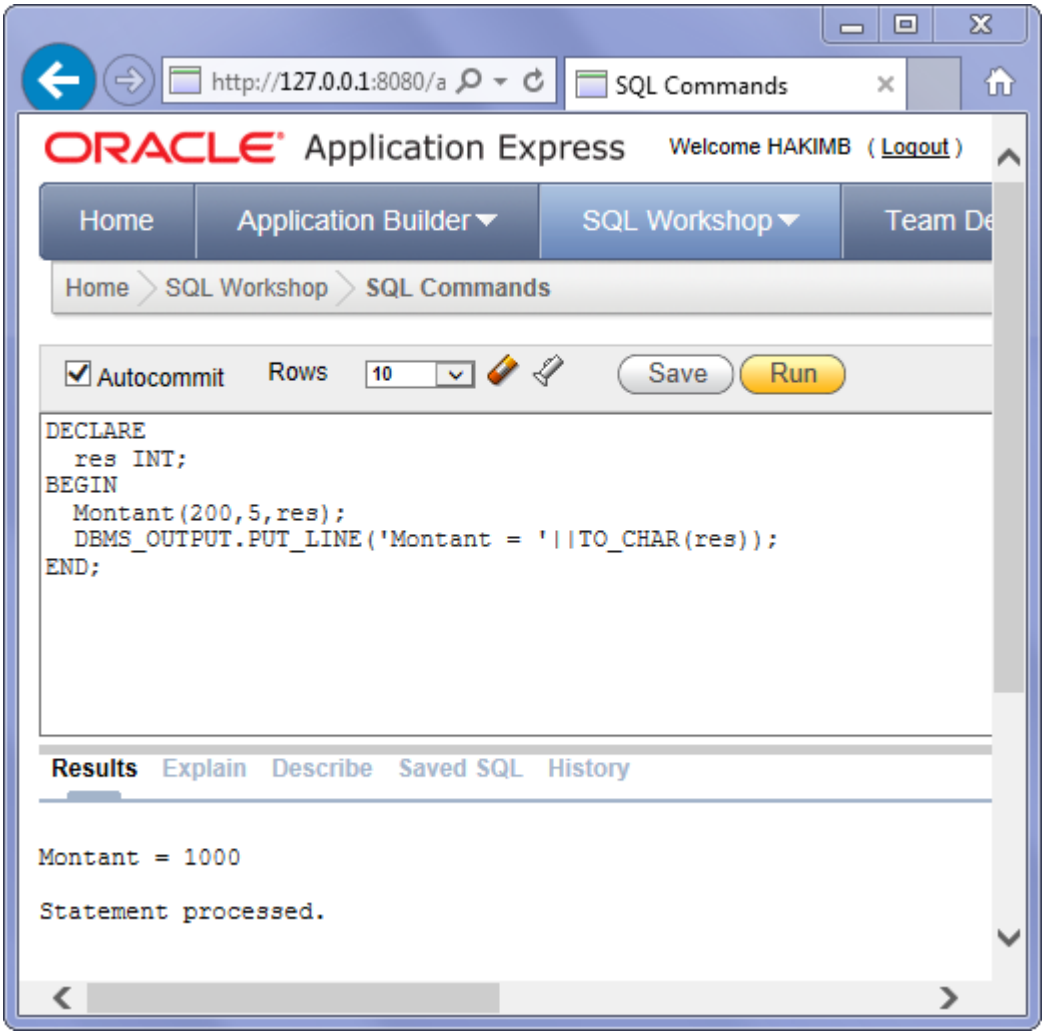


EXEMPLE3 (INPUT et OUTPUT)

Écrire une procédure qui calcul le montant à payer sachant le prix et la quantité.



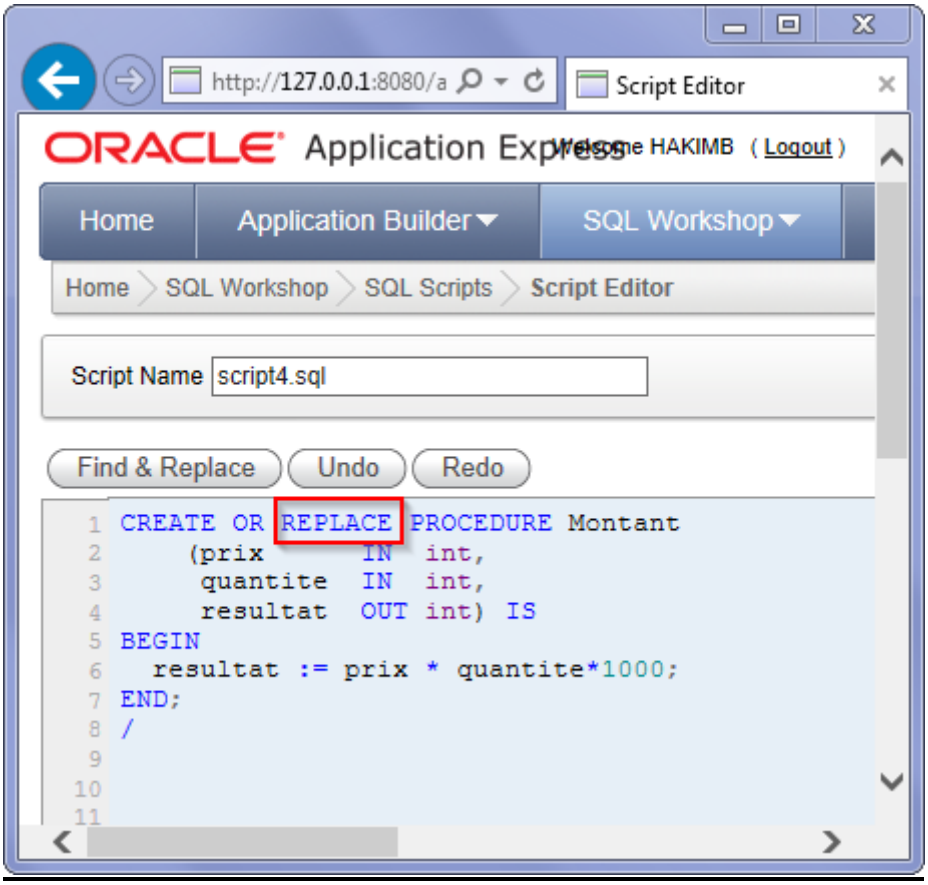
Exécuter la procédure :



4.4 Modification d'une procédure

La modification d'une procédure stockée se fait par l'instruction REPLACE PROCEDURE.

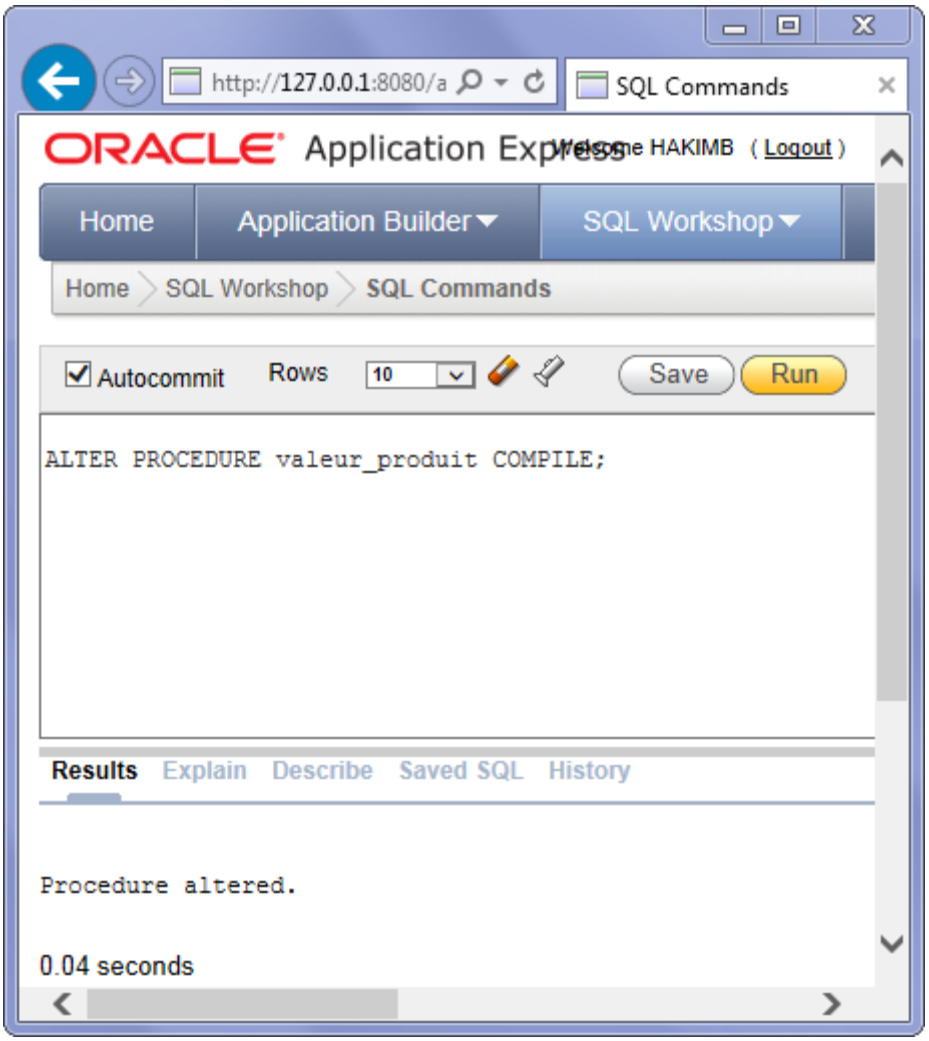
EXEMPLE



4.5 Recompiler une procédure

Une procédure peut devenir invalide. Par exemple si elle est basée sur une table dont la structure a changé.

Il est possible dans ce cas de recompiler une procédure avec la commande ALTER PROCEDURE RECOMPILE :

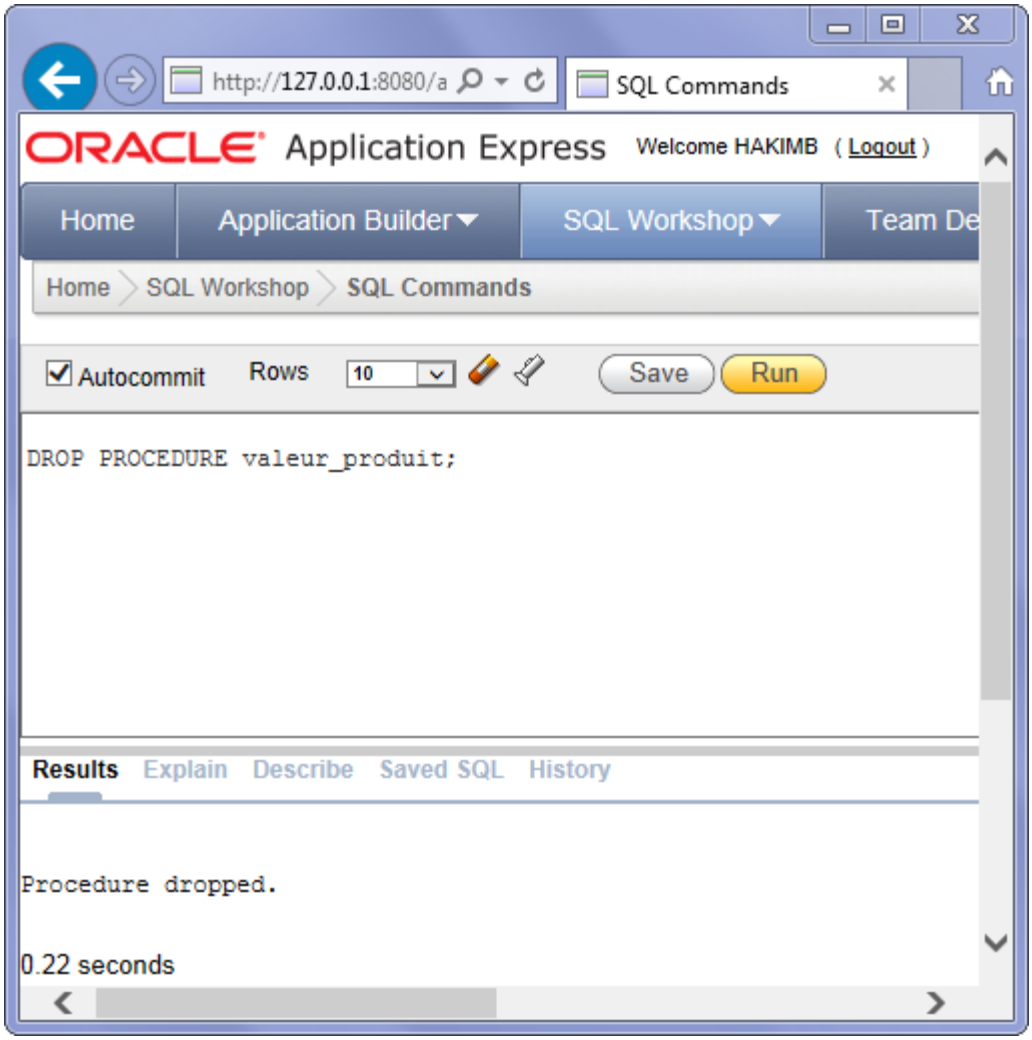


4.6 Supression d'une procédure

La modification d'une procédure stockée se fait par l'instruction DROP PROCEDURE.

EXEMPLE

Supprimer la procédure valeur_produit :



5 LES FONCTIONS

Une fonction est semblable à une procédure stockée à la différence qu'elle retourne toujours une valeur.

5.1 Création d'une fonction sclalaire

Les fonctions de type scalaire retournent, à l'aide du mot réservé RETURN, une valeur scalaire.

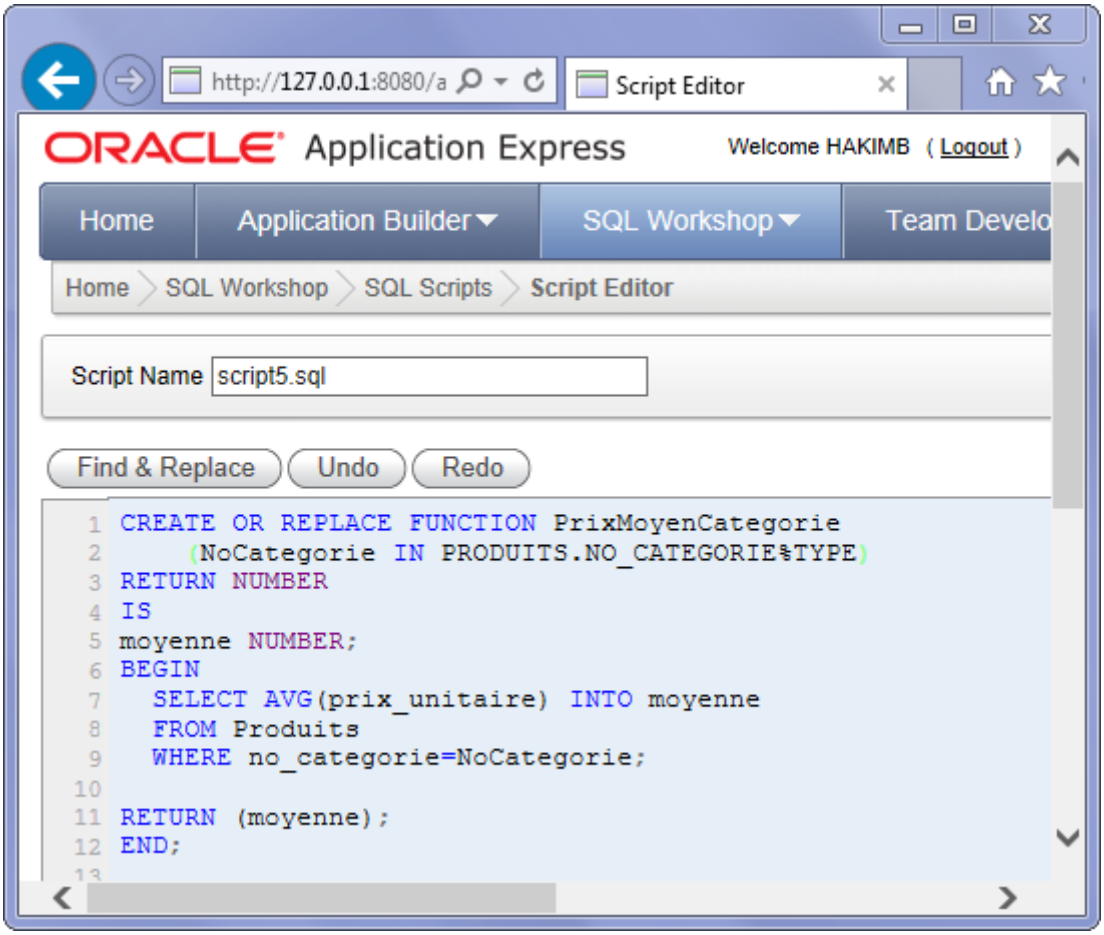
La syntaxe de l'instruction de création d'une fonction scalaire est la suivante :

```
CREATE [OR REPLACE] FUNCTION nom_fonction
   [ (paramètre [,paramètre]) ]
   RETURN type_de_données_de_la_valeur_retournée
IS | AS
   [Section déclaration]
BEGIN
    <CODE PL/SQL>
[EXCEPTION
   Section gestion des exceptions]

END [nom_fonction];
```

La fonction retourne une valeur du même type que la clause RETURN.

EXEMPLE

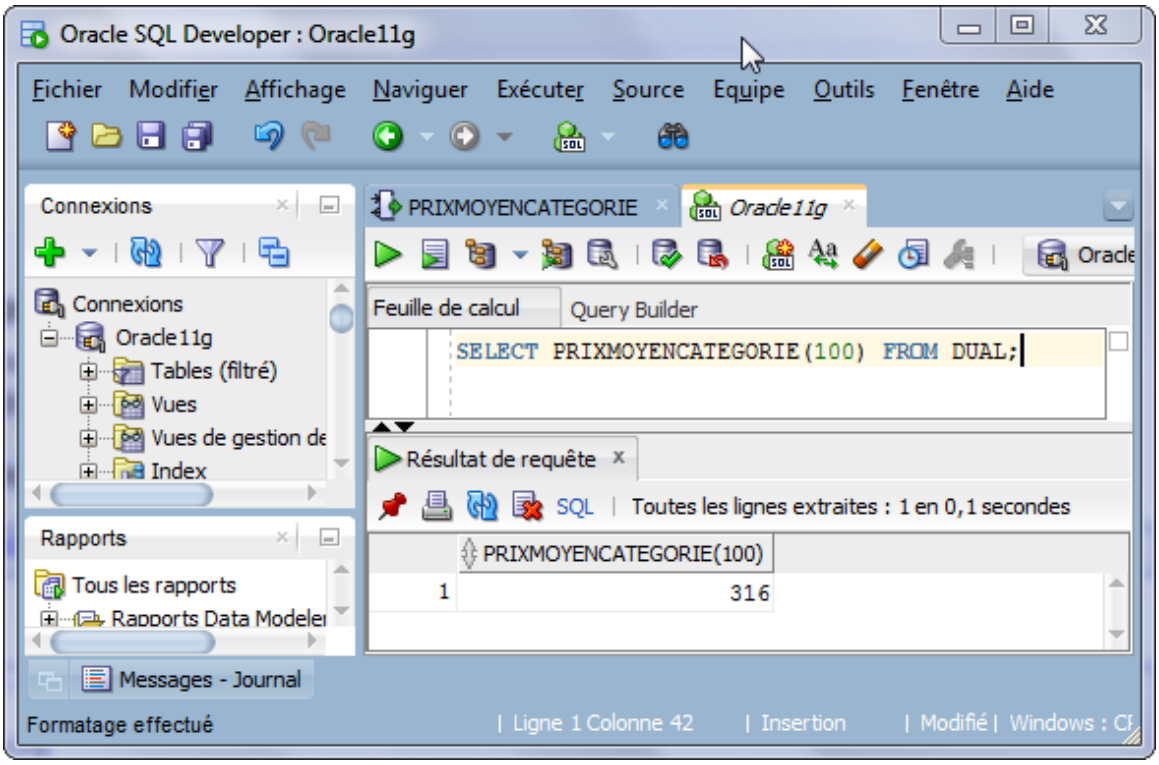
Créer une fonction qui retourne le prix moyen par catégorie de produits.



5.2 Appel d'une fonction scalaire

Une fonction s'utilise comme une procédure stockée, mais aussi comme une table.

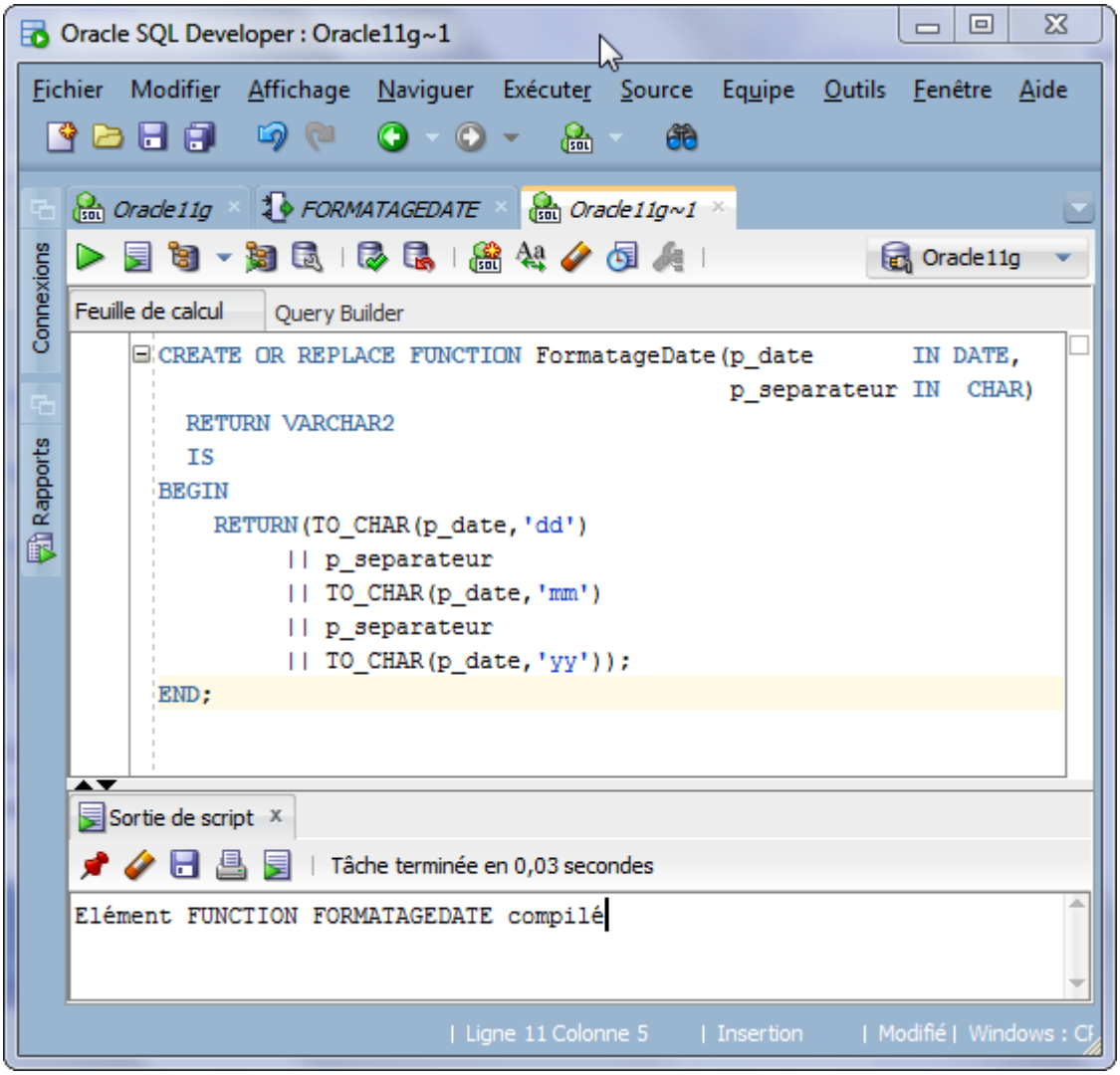
L'appel d'une fonction peut se faire avec l'instruction SELECT :



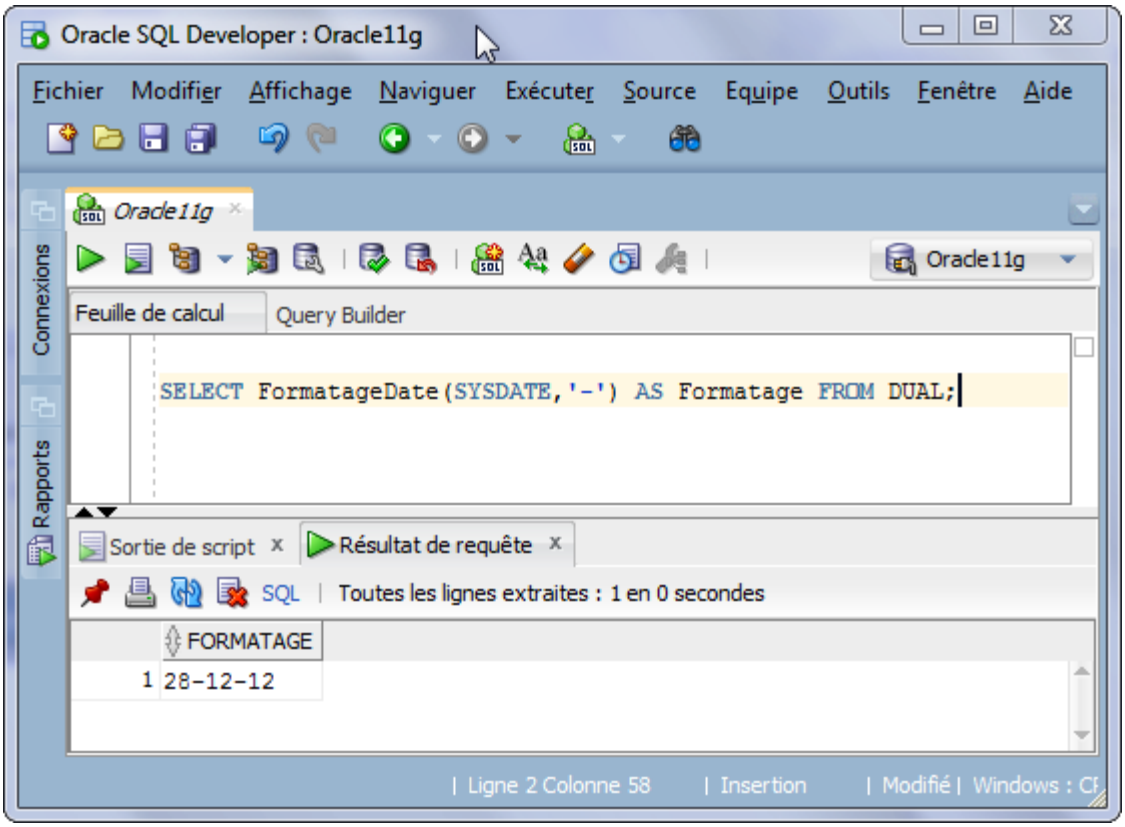
EXEMPLE

Écrire une fonction prend en paramètre une date et un séparateur et retourne la date dans le format jour mois année, séparés par le séparateur fourni en paramètre.

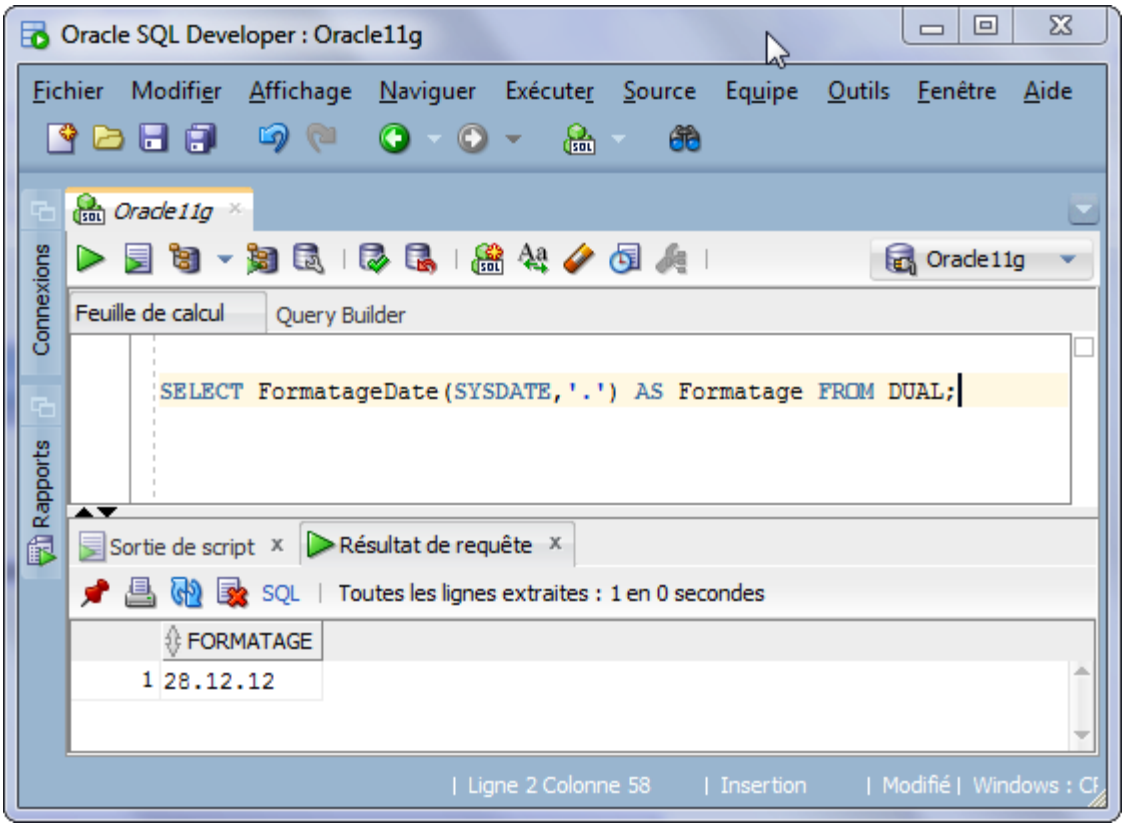
```
CREATE OR REPLACE FUNCTION FormatageDate(
          p_date       IN DATE,
          p_separateur IN  CHAR)
  RETURN VARCHAR2
  IS
BEGIN
    RETURN(TO_CHAR(p_date,'dd')
         || p_separateur
         || TO_CHAR(p_date,'mm')
         || p_separateur
         || TO_CHAR(p_date,'yy'));
END;
```


Oracle SQL Developer : Oracle11g~1
Fichier Modifier Affichage Naviguer Exécuter Source Equipe Outils Fenêtre Aide
Oracle11g
FORMATAGEDATE
Oracle11g~1
Connexions
Rapports
Oracle11g
Feuille de calcul
Query Builder
CREATE OR REPLACE FUNCTION FormatageDate(p_date IN DATE,
p_separateur IN CHAR)
RETURN VARCHAR2
IS
BEGIN
RETURN(TO_CHAR(p_date,'dd')
|| p_separateur
|| TO_CHAR(p_date,'mm')
|| p_separateur
|| TO_CHAR(p_date,'yy'));
END;
Sortie de script
Tâche terminée en 0,03 secondes
Elément FUNCTION FORMATAGEDATE compilé
Ligne 11 Colonne 5
Insertion
Modifié
Windows : CF

Appel de la fonction **FormatageDate** avec la date du jour (28 décembre 2012) et comme séparateur '-' :



Appel de la fonction **FormatageDate** avec la date du jour (28 décembre 2012) et comme séparateur '.' :

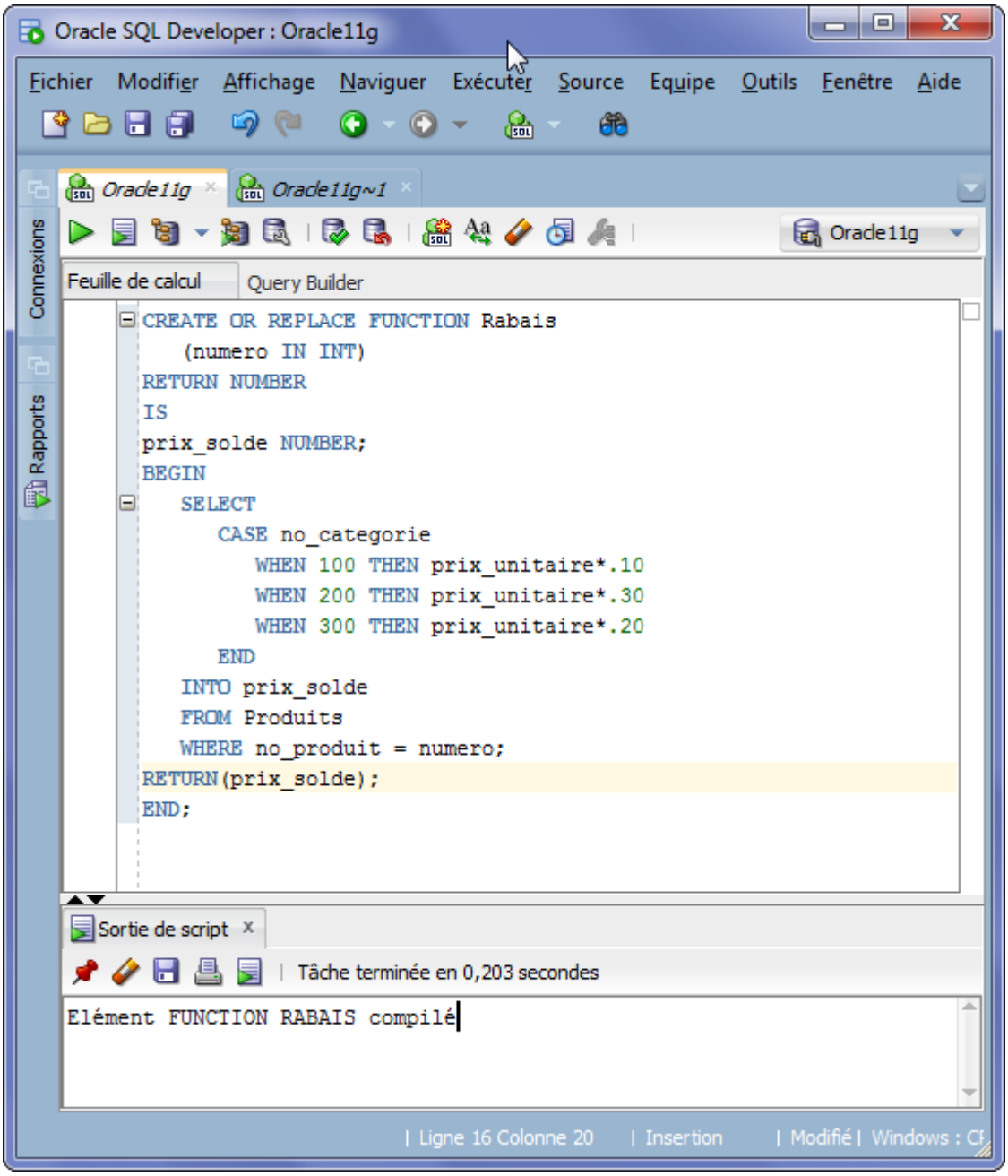


EXEMPLE

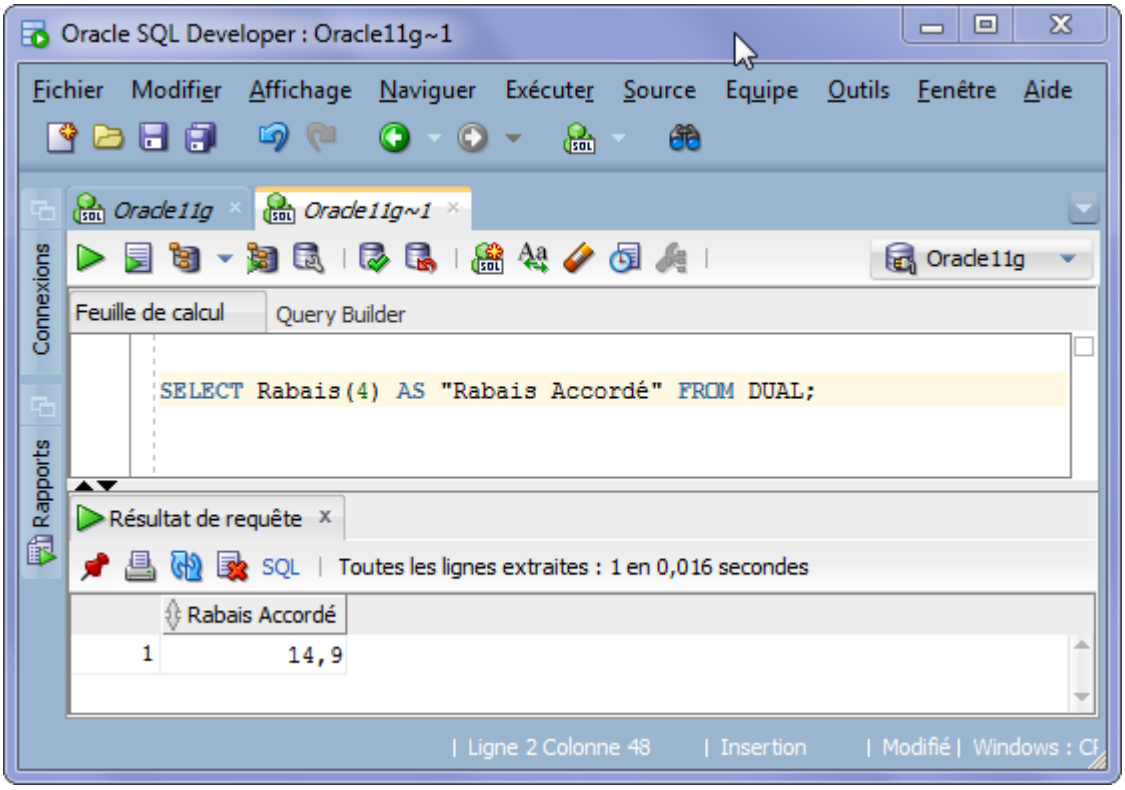
Écrire une fonction Rabais qui calcul le rabais accordé.

Un rabais est accordé selon la catégorie du produit comme suit :

- 10% de rabais sur les Tablettes
- 20% de rabais sur les Ordinateurs portables
- 30% de rabais sur les Cellulaires.



Appel de la fonction Rabais avec le produit dont le numéro est 4 :

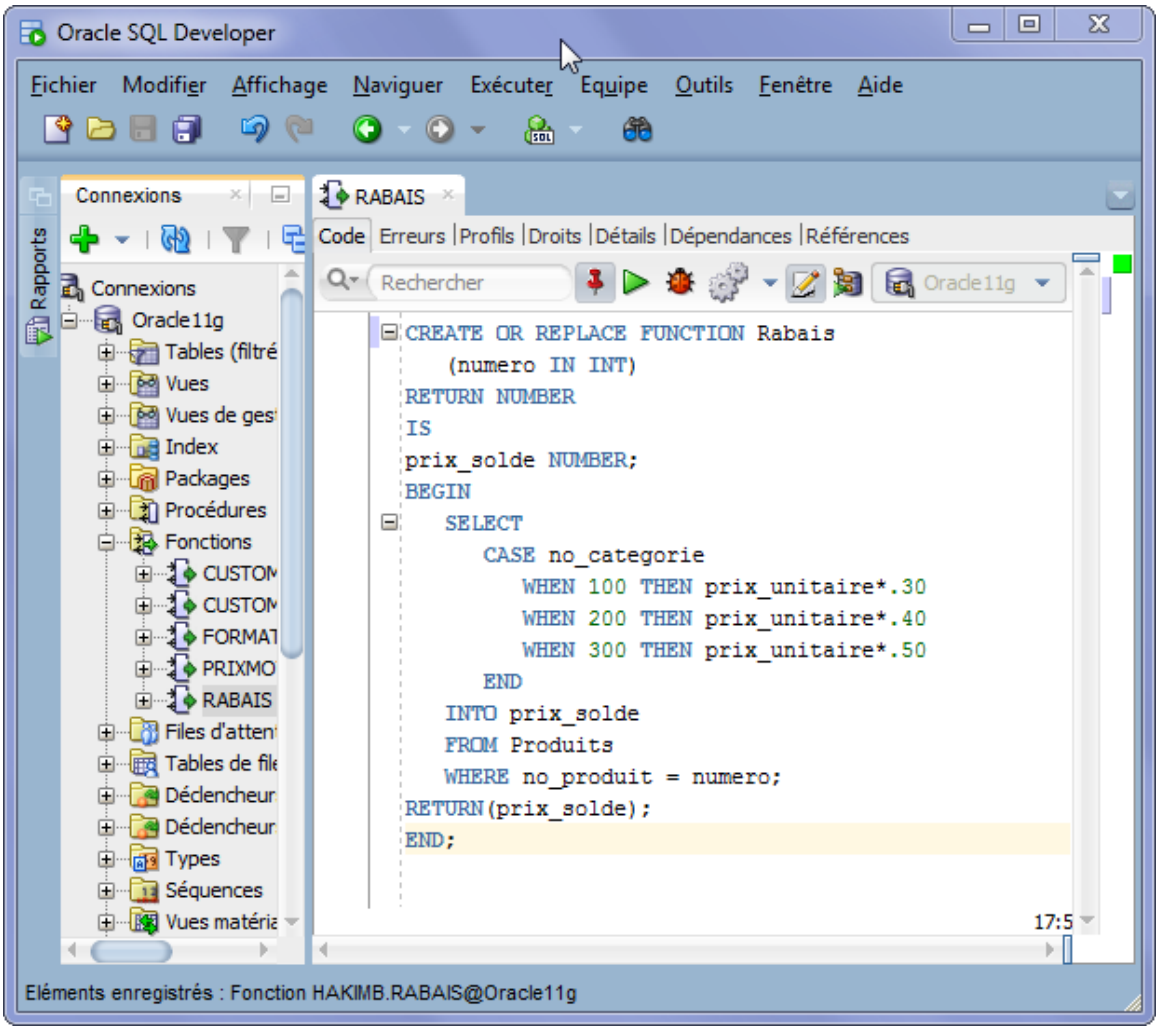


5.3 Modification d'une fonction

L'instruction **ALTER FUNCTION** permet de modifier une fonction.

EXEMPLE

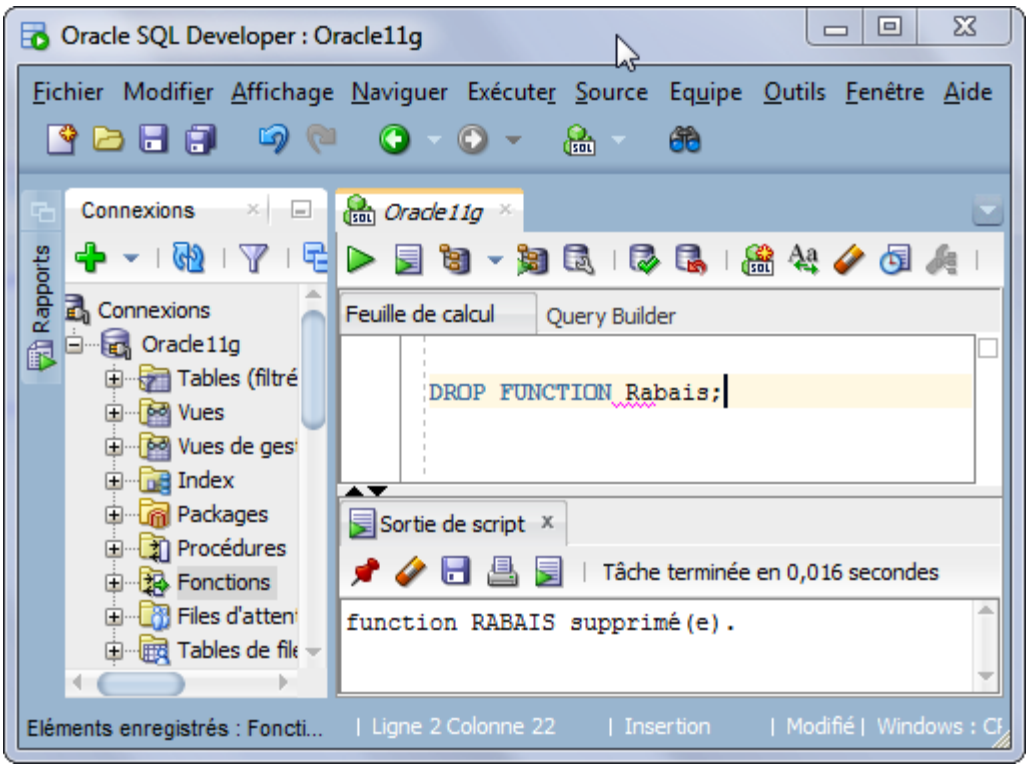
On va modifier la fonction Rabais pour changer la valeur du rabais :



5.4 Suppresion d'une fonction

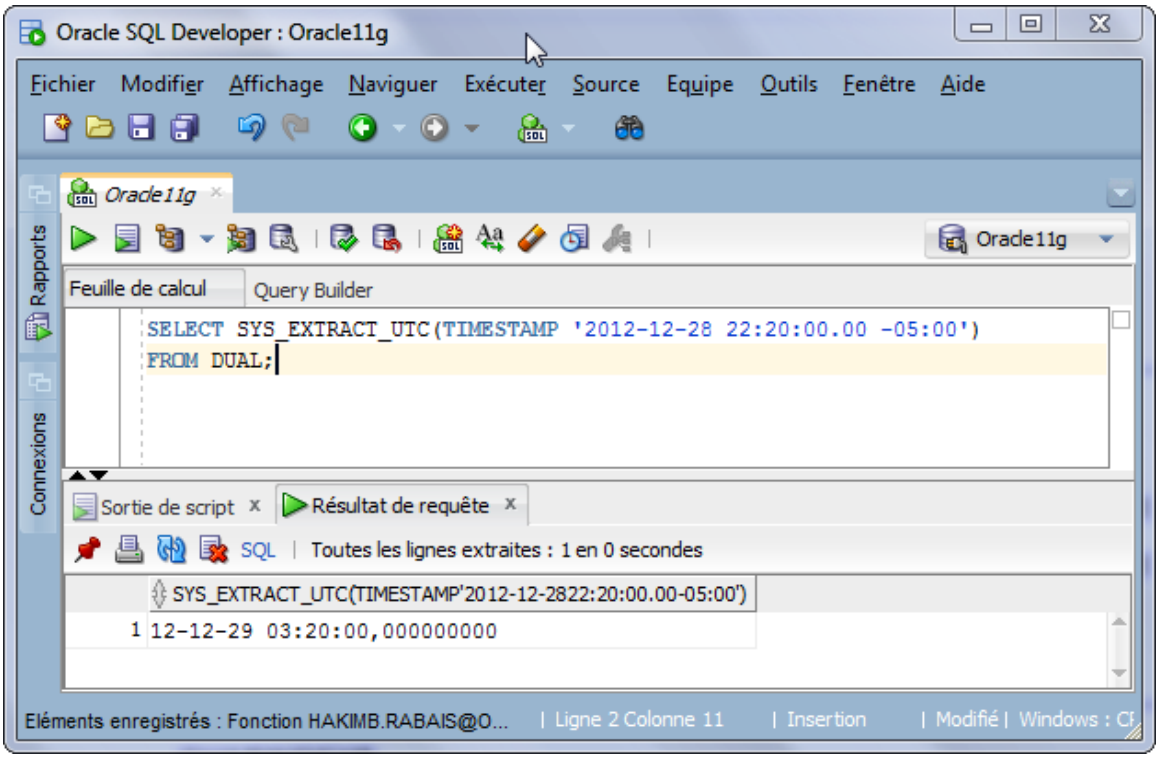
L'instruction **DROP FUNCTION** permet de supprimer une fonction.

EXEMPLE

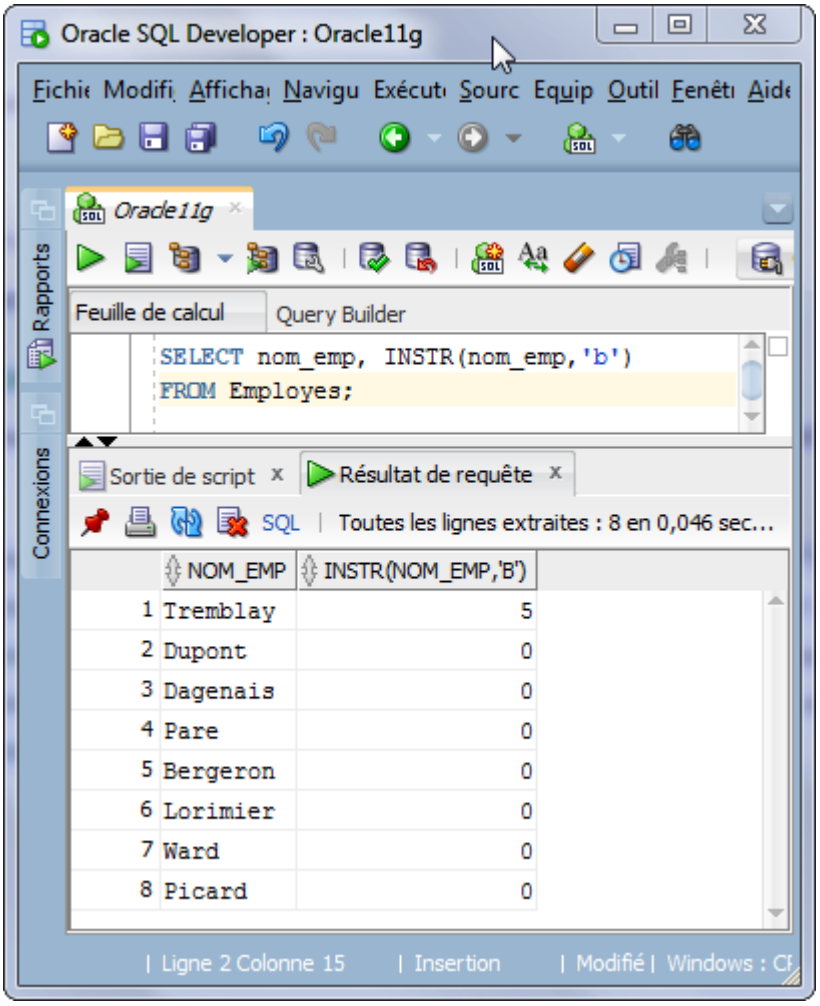


5.5 Les fonctions prédéfinis

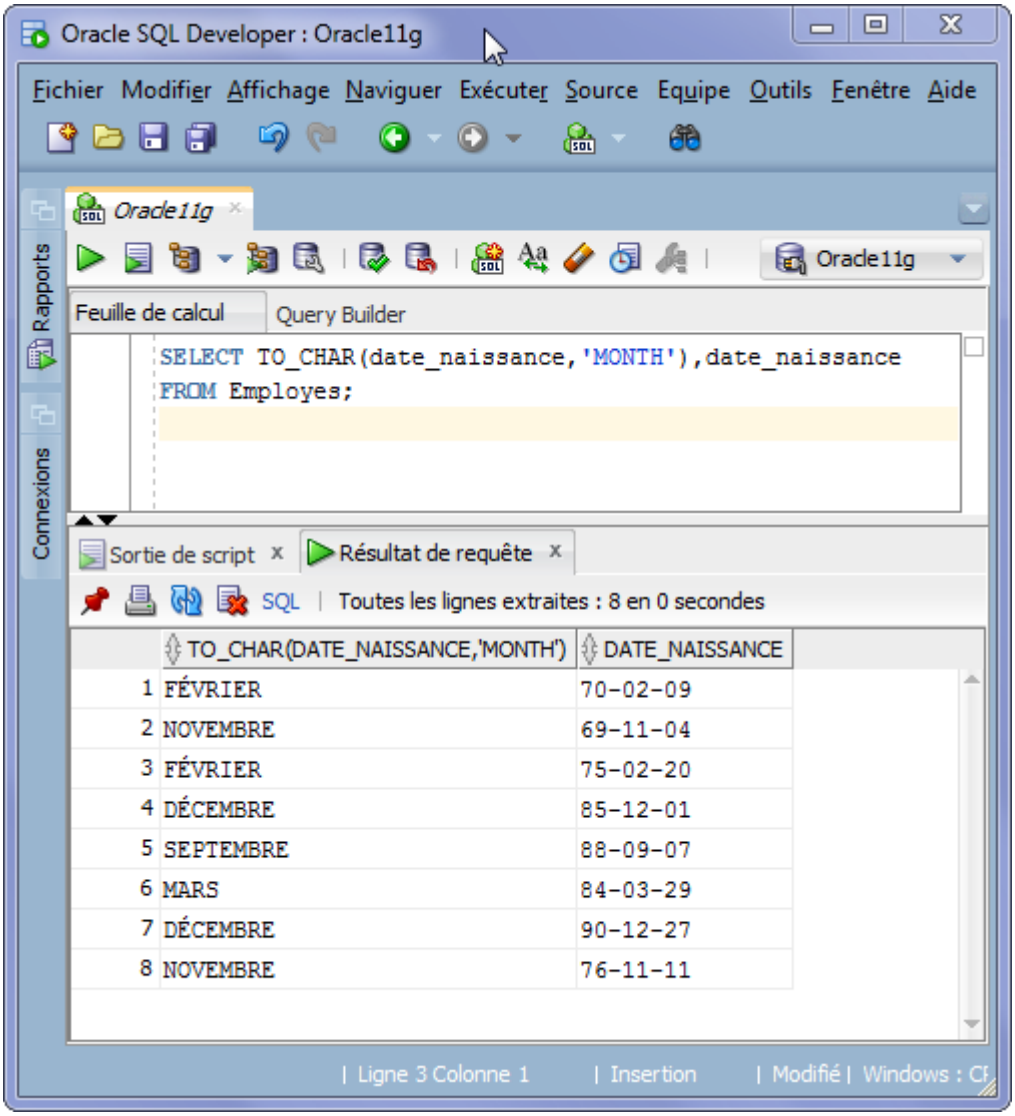
Fonction SYS_EXTRACT_UTC



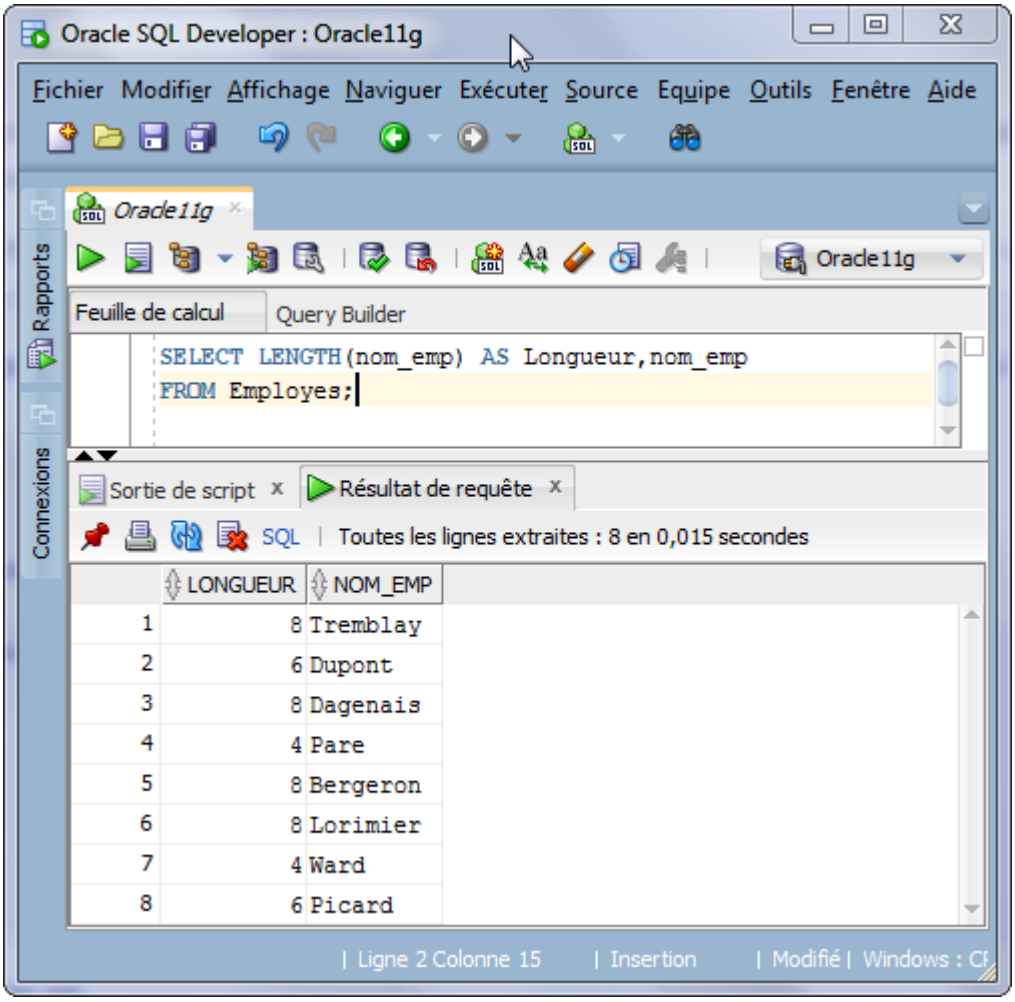
Fonction CHARINDEX



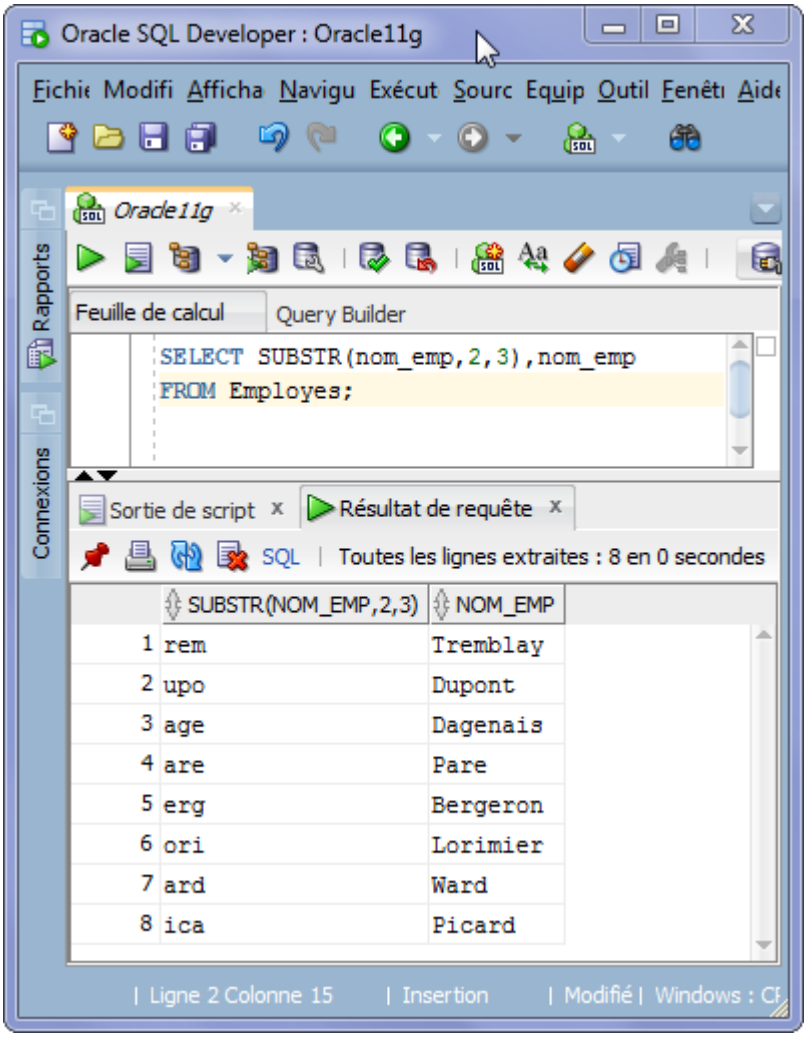
Fonction TO_CHAR



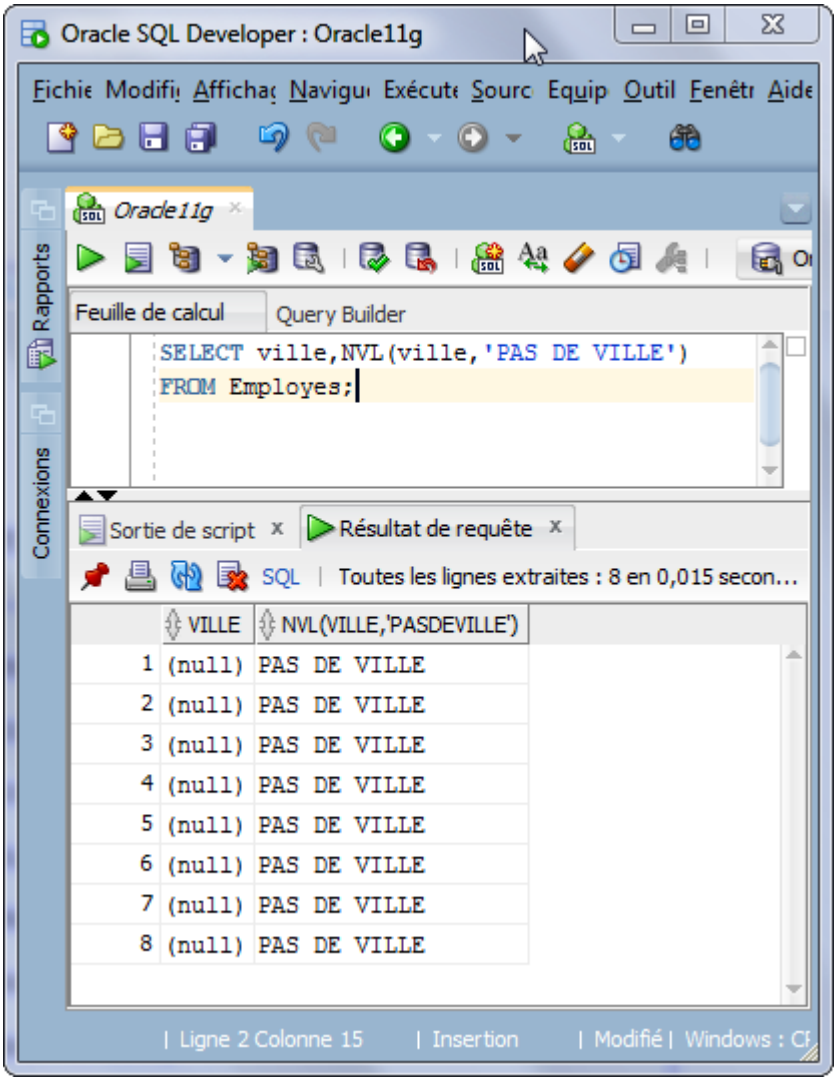
Fonction LENGTH



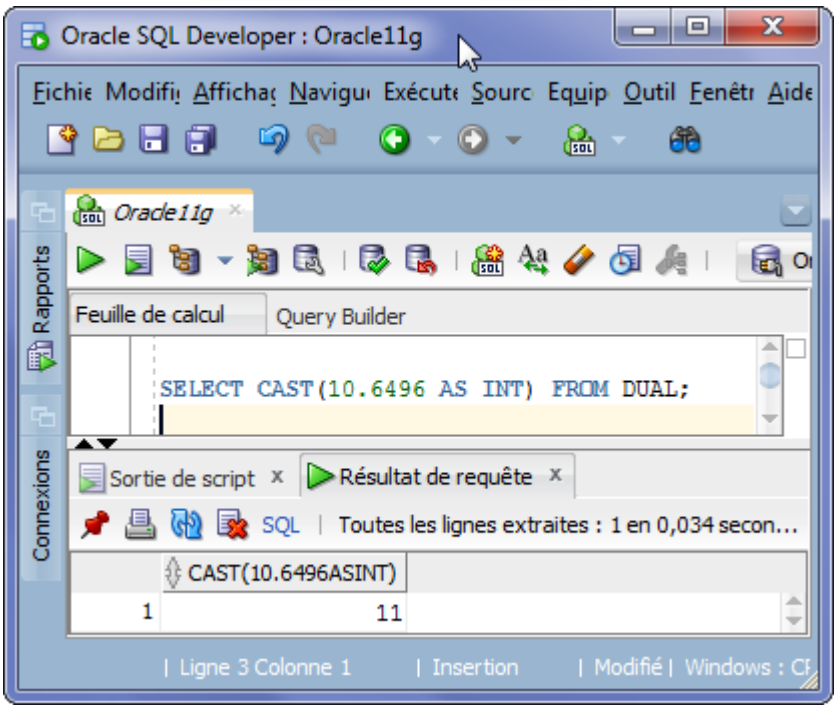
Fonction SUBSTR



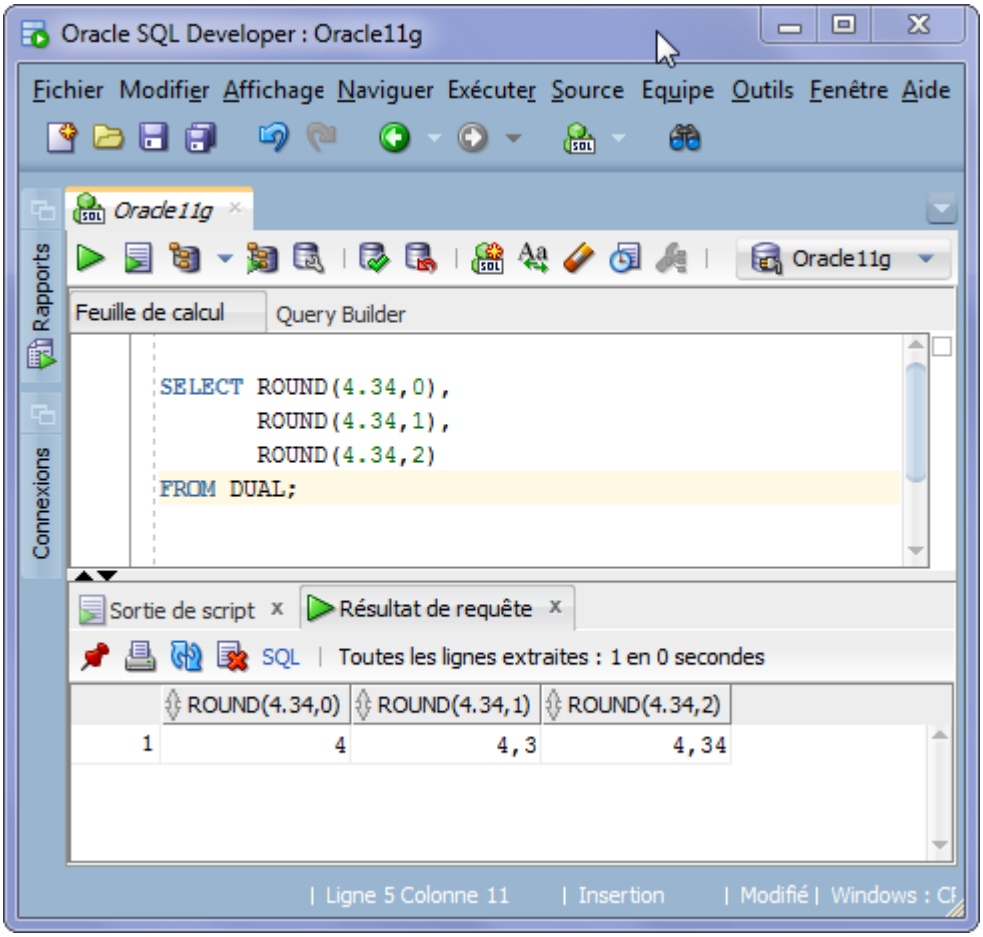
Fonction ISNULL



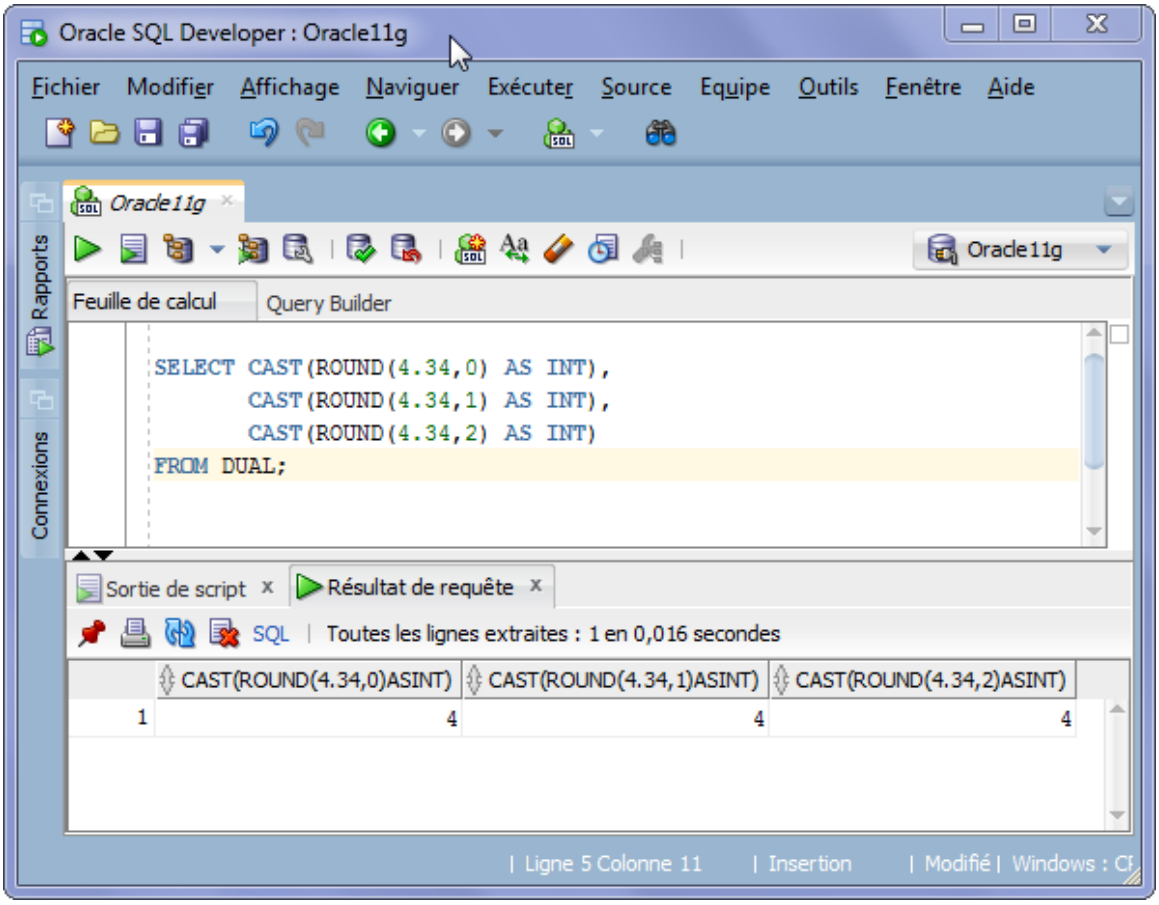
Fonction CAST

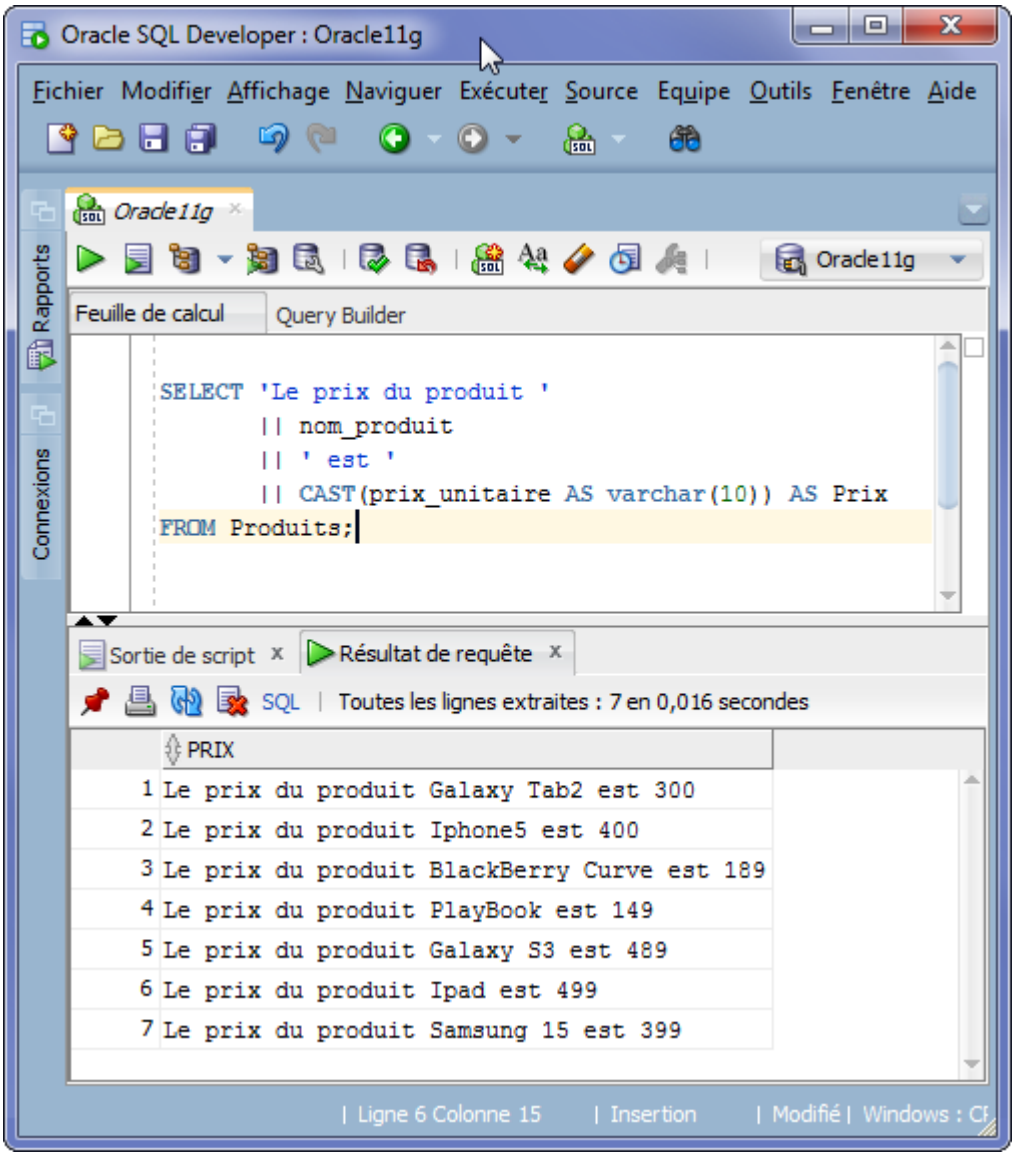


Fonction ROUND



Fonction CAST




Oracle SQL Developer : Oracle11g
Fichier Modifier Affichage Naviguer Exécuter Source Equipe Outils Fenêtre Aide
Oracle11g
Oracle11g
Feuille de calcul
Query Builder
SELECT 'Le prix du produit '
|| nom_produit
|| ' est '
|| CAST(prix_unitaire AS varchar(10)) AS Prix
FROM Produits;
Sortie de script
Résultat de requête
SQL | Toutes les lignes extraites : 7 en 0,016 secondes
PRIX
1 Le prix du produit Galaxy Tab2 est 300
2 Le prix du produit Iphone5 est 400
3 Le prix du produit BlackBerry Curve est 189
4 Le prix du produit PlayBook est 149
5 Le prix du produit Galaxy S3 est 489
6 Le prix du produit Ipad est 499
7 Le prix du produit Samsung 15 est 399
Ligne 6 Colonne 15
Insertion
Modifié
Windows : Cf
Rapports
Connexions

6 LES PACKAGE PL/SQL

Souvent on besoin d'écrire plusieurs procédures et fonctions pour répondre à nos besoin. Par exemple dans le domaine comptable on doit avoir une procédure ou fonction pour chaque tâche spécifique. Dans ce cas il serait intéressant de regrouper ces fonctions et procédures dans un paquetage.

La création d'un **PACKAGE PL/SQL** se fait en deux étapes :

6.1 PACKAGE SPECIFICATION

Dans le **PACKAGE SPCIFICATION** on déclare toutes les procédures et fonctions qui font partie du package ainsi que leurs paramètres.

La création du **PACKAGE SPCIFICATION** se fait comme suit :

```
CREATE OR REPLACE  PACKAGE nom_du_package  AS
 FUNCTION nom_fonction(paramètres de la fonction)
 RETURN type de retour;
...
 PROCEDURE soutraction(paramètres de la procedure);
...

END nom_du_package;
END OPERATION;
```

6.2 PACKAGE BODY

Dans le **PACKAGE BODY** on retrouve la définition de chaque procédure et fonction du package.

REMARQUE

Lors de l'exécution d'une procédure ou d'une fonction d'un package, on doit préfixer la procédure ou le package par le nom du package.

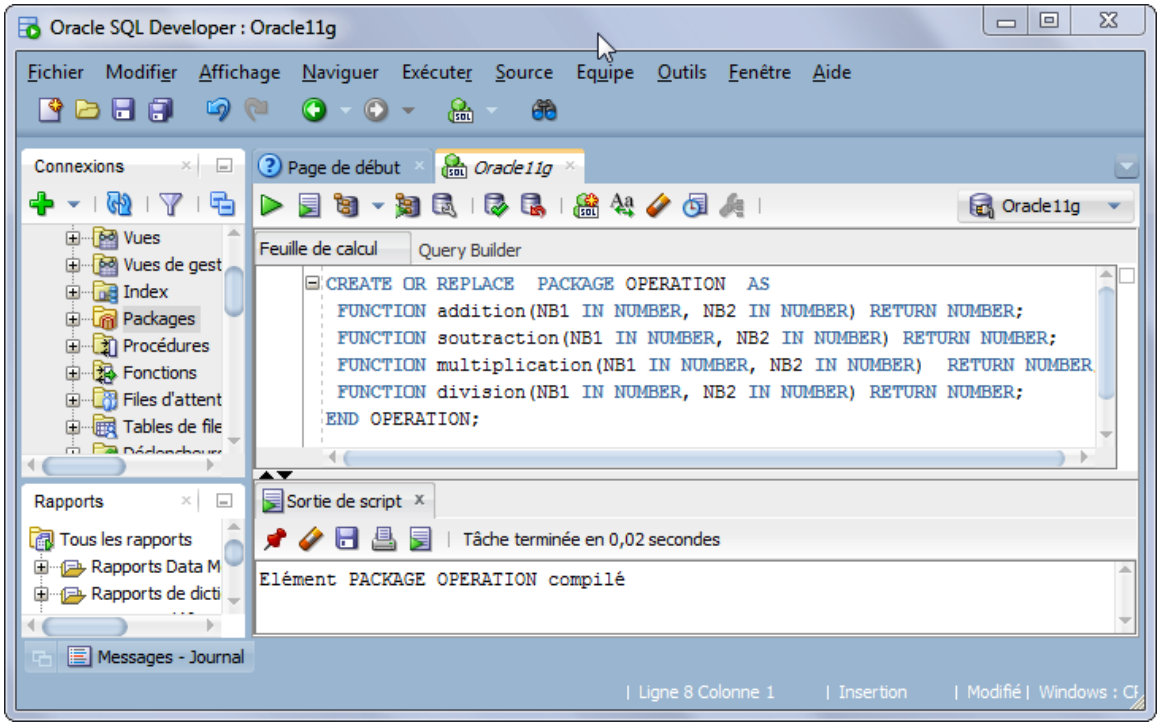
```
Nom_package.nom_procedure

Nom_package.nom_fonction
```

6.3 EXEMPLE PRATIQUE

Écrire un **package** PL/SQL nommé **OPERATION** qui permet de réaliser les opérations arithmétiques :

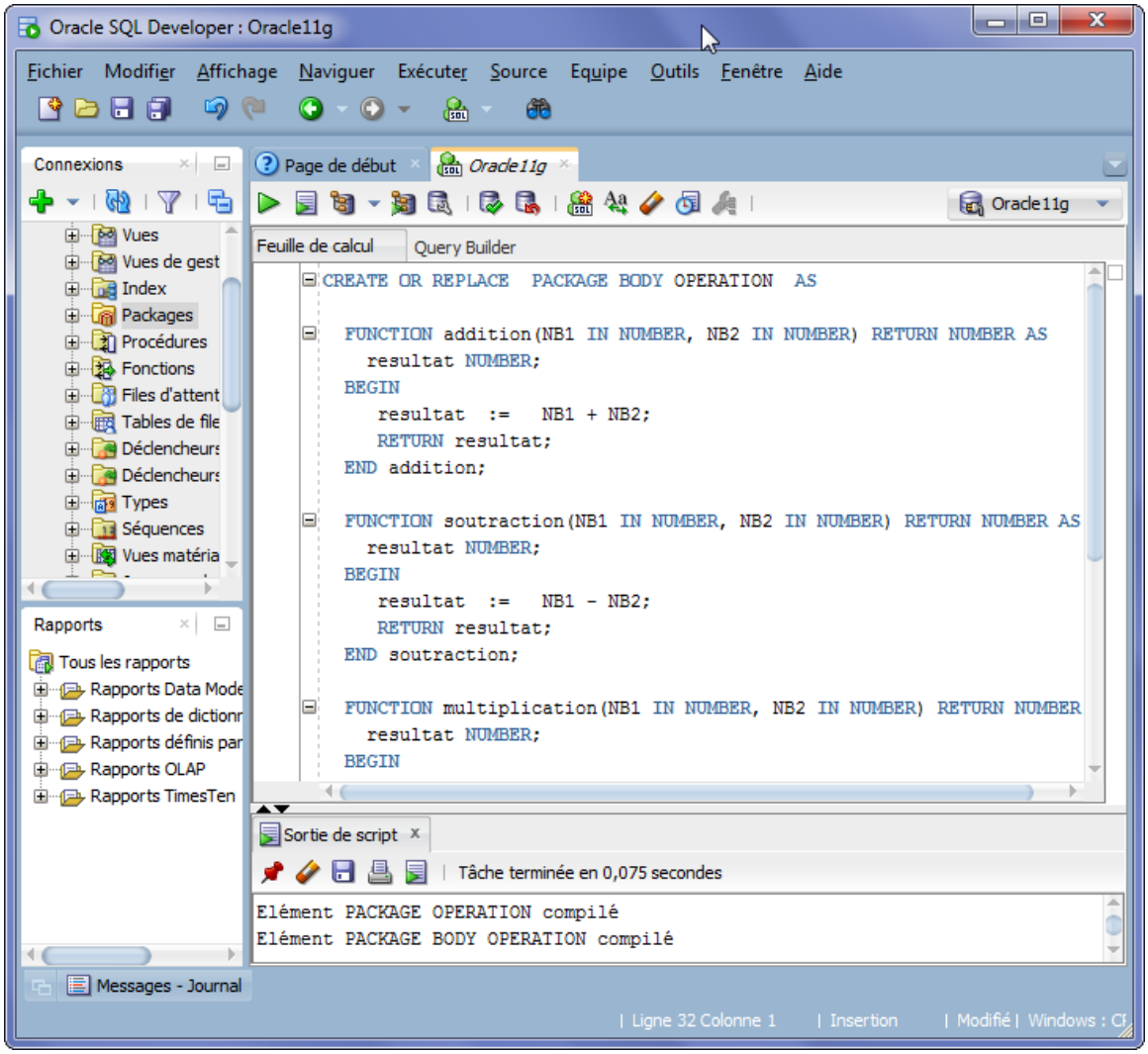
1) En premier il faut créer le **PACKAGE** en déclarant toutes les procédures et les fonctions :

```
CREATE OR REPLACE  PACKAGE OPERATION  AS
 FUNCTION addition(NB1 IN NUMBER, NB2 IN NUMBER)
 RETURN NUMBER;

 FUNCTION soutraction(NB1 IN NUMBER, NB2 IN NUMBER)
 RETURN NUMBER;

 FUNCTION multiplication(NB1 IN NUMBER, NB2 IN NUMBER)
 RETURN NUMBER;

 FUNCTION division(NB1 IN NUMBER, NB2 IN NUMBER)
 RETURN NUMBER;

END OPERATION;
```

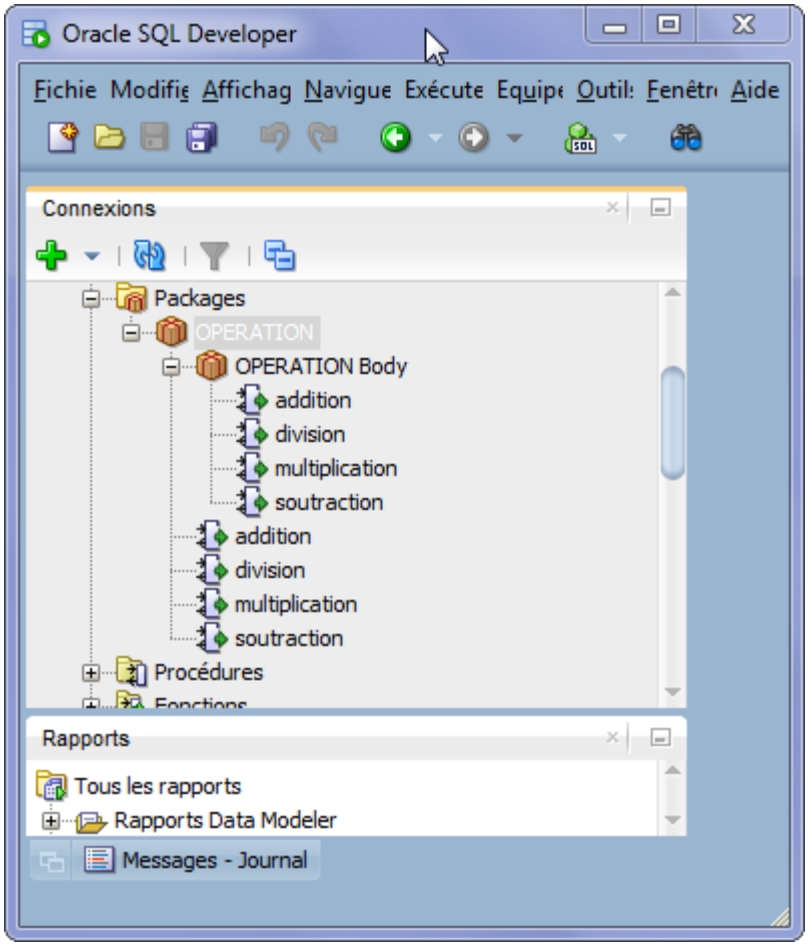


2) Ensuite on doit créer le **PACKAGE BODY** dans lequel on retrouve la définition de chaque procédure et fonction :

```
CREATE OR REPLACE  PACKAGE BODY OPERATION  AS

  FUNCTION addition(NB1 IN NUMBER, NB2 IN NUMBER)
  RETURN NUMBER AS
    resultat NUMBER;
  BEGIN
     resultat  :=  NB1 + NB2;
     RETURN resultat;
  END addition;

  FUNCTION soutraction(NB1 IN NUMBER, NB2 IN NUMBER)
  RETURN NUMBER AS
    resultat NUMBER;
  BEGIN
     resultat  :=  NB1 - NB2;
     RETURN resultat;
  END soutraction;

  FUNCTION multiplication(NB1 IN NUMBER, NB2 IN
  NUMBER) RETURN NUMBER AS
    resultat NUMBER;
  BEGIN
     resultat  :=  NB1 * NB2;
     RETURN resultat;
  END multiplication;

  FUNCTION division(NB1 IN NUMBER, NB2 IN NUMBER)
  RETURN NUMBER AS
    resultat NUMBER;
  BEGIN
     resultat  :=  NB1 / NB2;
     RETURN resultat;
  END division;

END OPERATION;
```

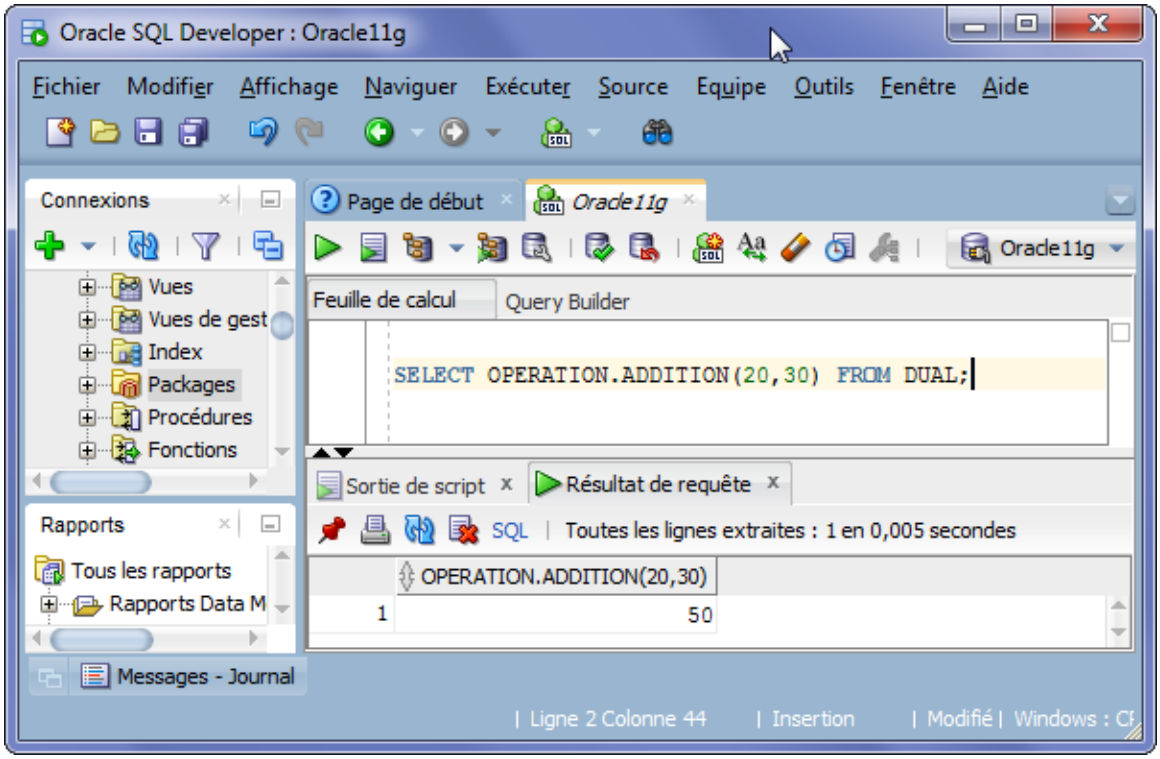

Oracle SQL Developer : Oracle11g
Fichier Modifier Affichage Naviguer Exécuter Source Equipe Outils Fenêtre Aide
Connexions
Vues
Vues de gest
Index
Packages
Procédures
Fonctions
Files d'attent
Tables de file
Déclencheurs
Déclencheurs
Types
Séquences
Vues matéria
Rapports
Tous les rapports
Rapports Data Mode
Rapports de dictionr
Rapports définis par
Rapports OLAP
Rapports TimesTen
Page de début
Oracle11g
Oracle11g
Feuille de calcul
Query Builder
CREATE OR REPLACE PACKAGE BODY OPERATION AS
FUNCTION addition(NB1 IN NUMBER, NB2 IN NUMBER) RETURN NUMBER AS
resultat NUMBER;
BEGIN
resultat := NB1 + NB2;
RETURN resultat;
END addition;
FUNCTION soutraction(NB1 IN NUMBER, NB2 IN NUMBER) RETURN NUMBER AS
resultat NUMBER;
BEGIN
resultat := NB1 - NB2;
RETURN resultat;
END soutraction;
FUNCTION multiplication(NB1 IN NUMBER, NB2 IN NUMBER) RETURN NUMBER
resultat NUMBER;
BEGIN
Sortie de script
Tâche terminée en 0,075 secondes
Elément PACKAGE OPERATION compilé
Elément PACKAGE BODY OPERATION compilé
Messages - Journal
Ligne 32 Colonne 1
Insertion
Modifié
Windows : Cr

Le package est bien créé dans la base de données :



3) Et pour terminer, exécuter une fonction ou procédure du **PACKAGE :**

```
SELECT OPERATION.ADDITION(20,30) FROM DUAL;
```



7 LES VUES

Une vue peut être considérée comme une requête enregistrée. Ainsi vous pouvez réutiliser une instruction sans avoir à la redéfinir.

Une vue se comporte comme table à quelques restrictions près, mais n'occupant pas d'espace disque pour les données. Une vue ne contient que la définition de la requête SQL.

Avantage des vues

Une vue vous permet d'effectuer les tâches suivantes :

- limiter l'accès d'un utilisateur à certaines lignes d'une table ;
- limiter l'accès d'un utilisateur à certaines colonnes d'une table ;
- joindre les colonnes de différentes tables ;
- une vue peut être utilisée partout où on peut utiliser une table ;

7.1 Création d'une vue

On peut créer une vue en utilisant le langage de définition de données .

La commande **CREATE VIEW** permet de créer une vue en spécifiant le SELECT constituant la définition de la vue :

```
CREATE VIEW nom_vue [(nom_col1,...)]
[WITH SCHEMABINDING]
AS
Instruction SELECT [WITH CHECK OPTION];
```

La spécification des noms de colonnes de la vue est facultative. Par défaut, les noms des colonnes de la vue sont les mêmes que les noms des colonnes du SELECT.

Si certaines colonnes du SELECT sont des expressions, il faut renommer ces colonnes dans le SELECT, ou spécifier les noms de colonne de la vue.

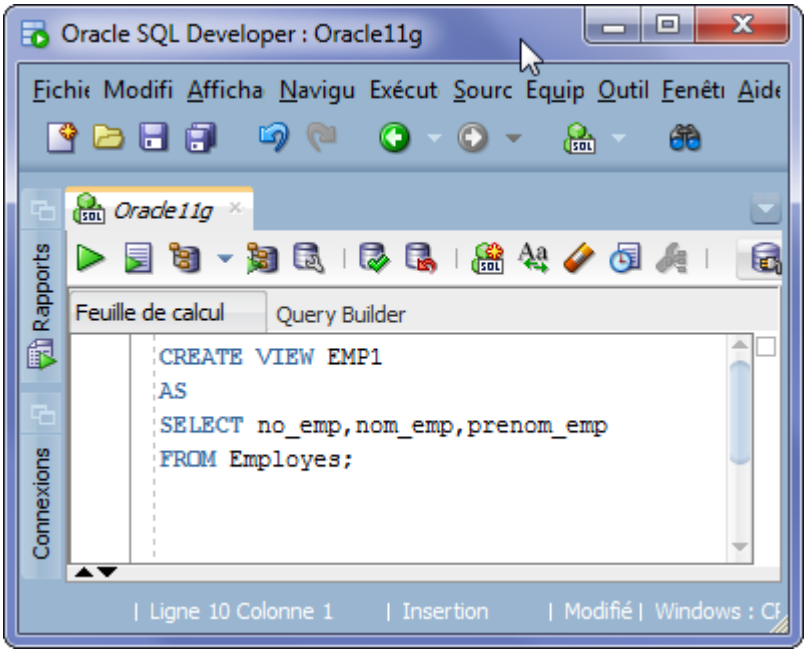
Une fois créée, une vue s'utilise comme une table. Il n'y a pas de duplication des données.

Le CHECK OPTION permet de vérifier que la mise à jour ou l'insertion faite à travers la vue ne produisent que des lignes qui font partie de la sélection de la vue.

La clause WITH SCHEMABINDING permet d'empêcher la modification ou la suppression des objets référencés par la vue.

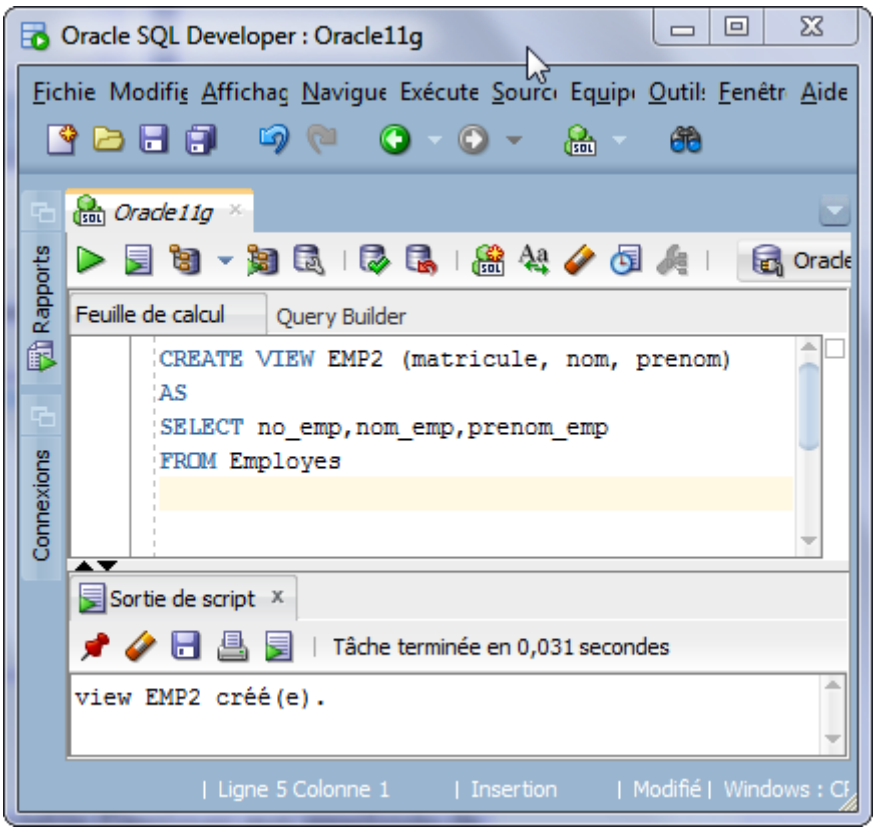
EXEMPLE1

Créer une vue sans renommer les colonnes :



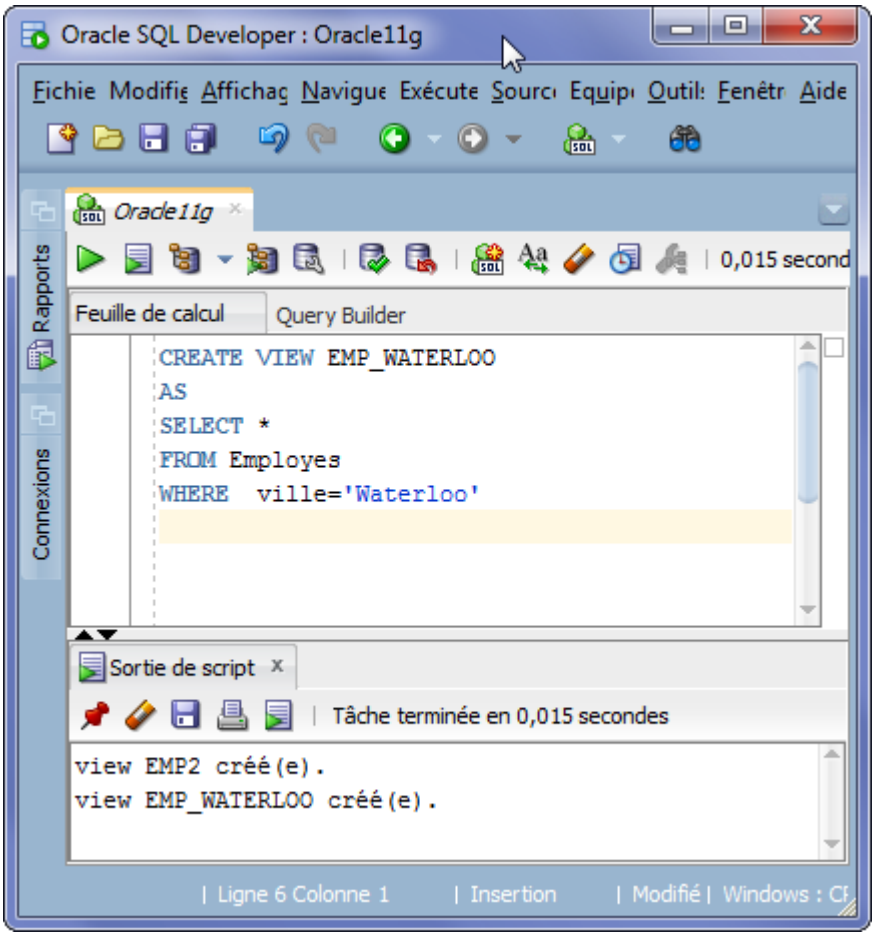
EXEMPLE2

On peut aussi créer une vue en renommant les colonnes :

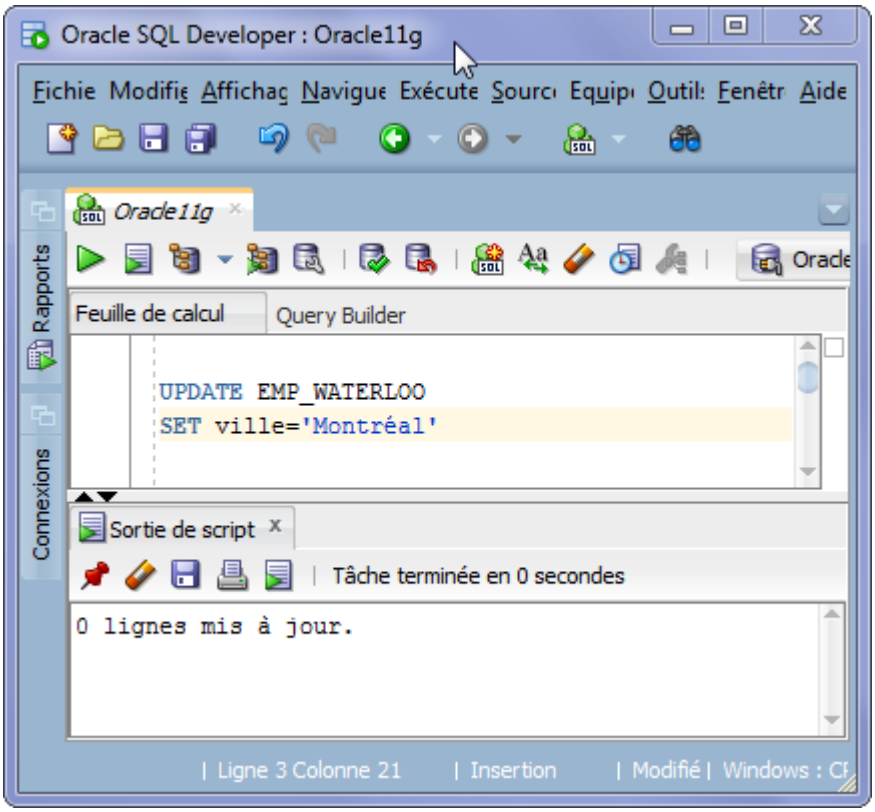

Oracle SQL Developer : Oracle11g
Fichie Modifie Affichag Navigue Exécute Sourc Equip Outil: Fenêtr Aide
Oracle11g
Rapports
Connexions
Feuille de calcul
Query Builder
CREATE VIEW EMP2 (matricule, nom, prenom)
AS
SELECT no_emp,nom_emp,prenom_emp
FROM Employes
Sortie de script
Tâche terminée en 0,031 secondes
view EMP2 créé(e).
Ligne 5 Colonne 1
Insertion
Modifié
Windows : Cr

EXEMPLE3

Création d'une vue constituant une restriction de la table *Employes* aux employés de la ville de *Waterloo*:

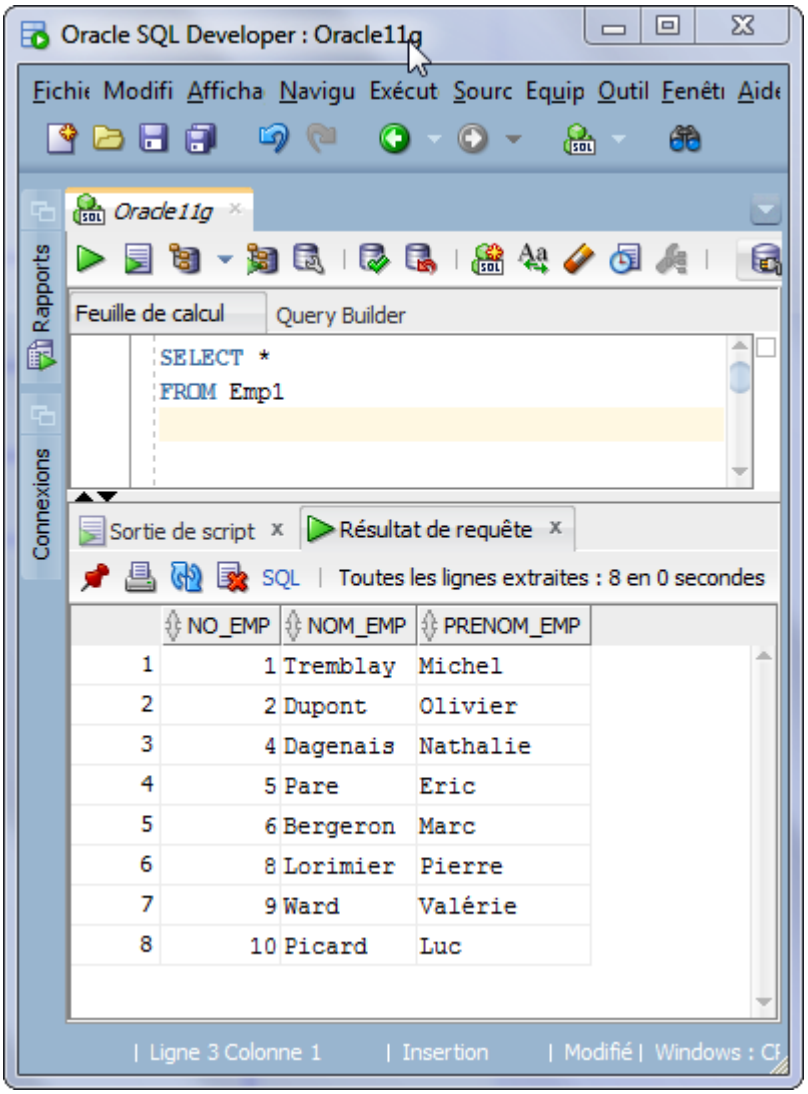


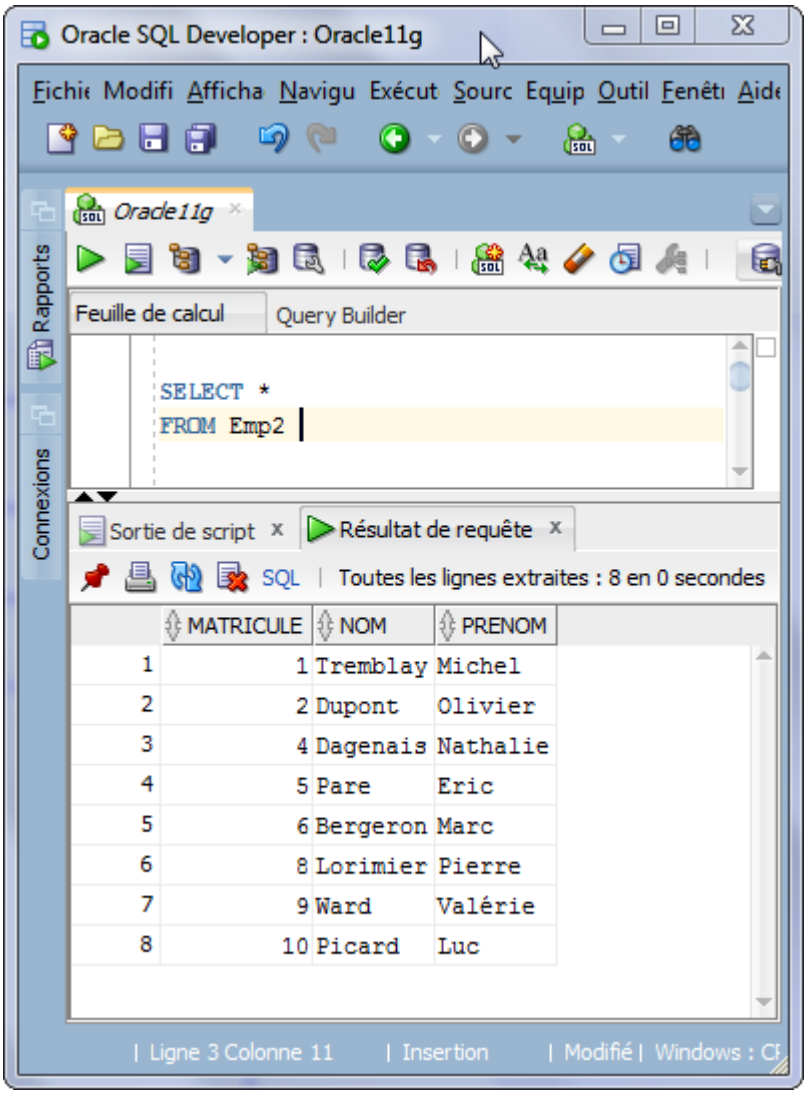
L’ordre SQL suivant, qui tente de mettre à jour la ville des employés de *Waterloo*.



7.2 Utiliser une vue

Une vue s'utilise comme un table.



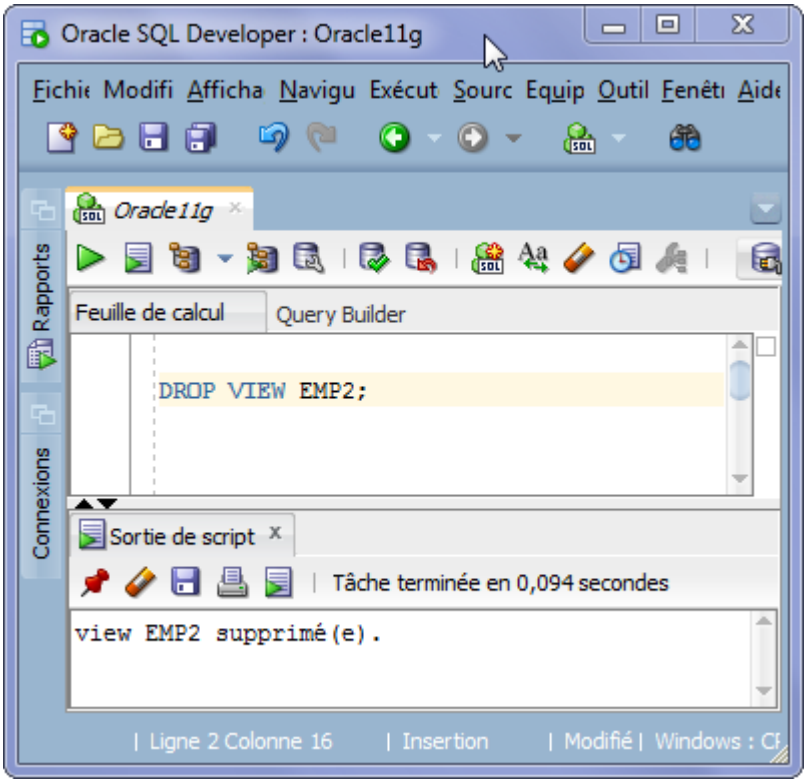

Oracle SQL Developer : Oracle11g
Fichie Modifi Afficha Navigu Exécut Sourc Equip Outil Fenêtı Aide
Oracle11g
Rapports
Connexions
Feuille de calcul
Query Builder
SELECT *
FROM Emp2
Sortie de script
Résultat de requête
SQL | Toutes les lignes extraites : 8 en 0 secondes
MATRICULE NOM PRENOM
1 1 Tremblay Michel
2 2 Dupont Olivier
3 4 Dagenais Nathalie
4 5 Pare Eric
5 6 Bergeron Marc
6 8 Lorimier Pierre
7 9 Ward Valérie
8 10 Picard Luc
Ligne 3 Colonne 11
Insertion
Modifié
Windows : Cf

7.3 Suppression d'une vue

L'instruction **DROP VIEW** permet de supprimer une vue.

```
DROP  VIEW nom_vue;
```

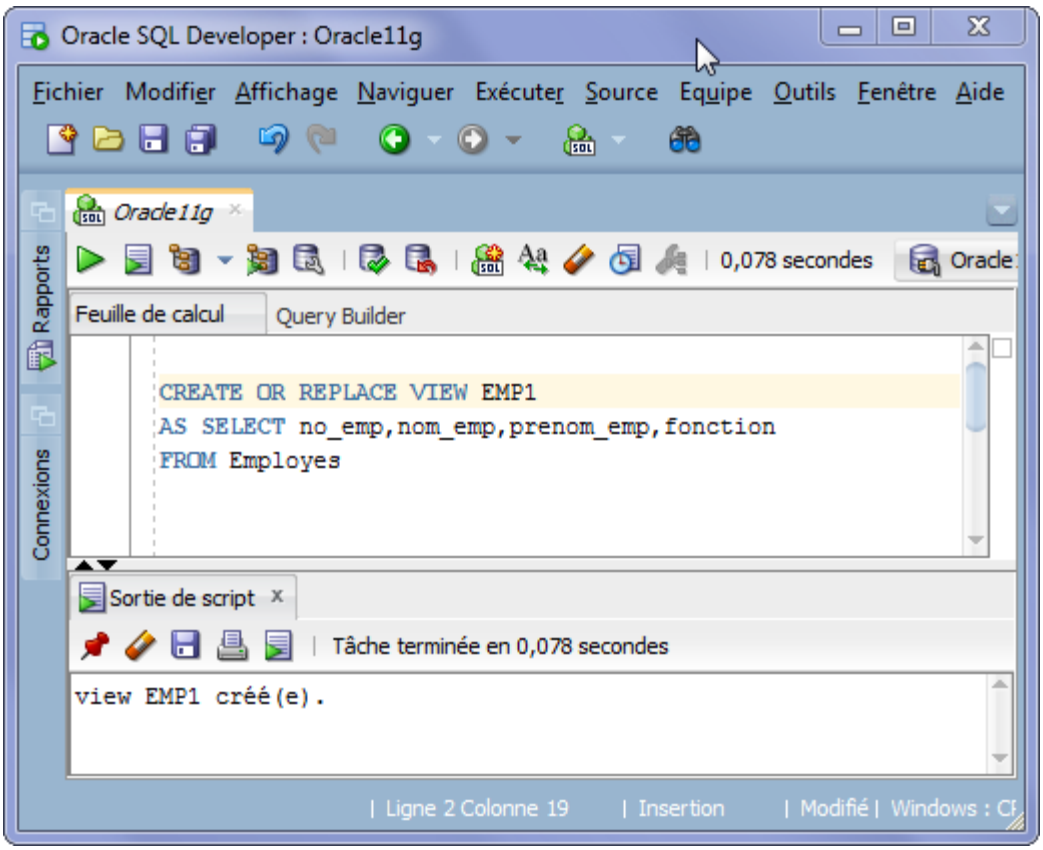
EXEMPLE



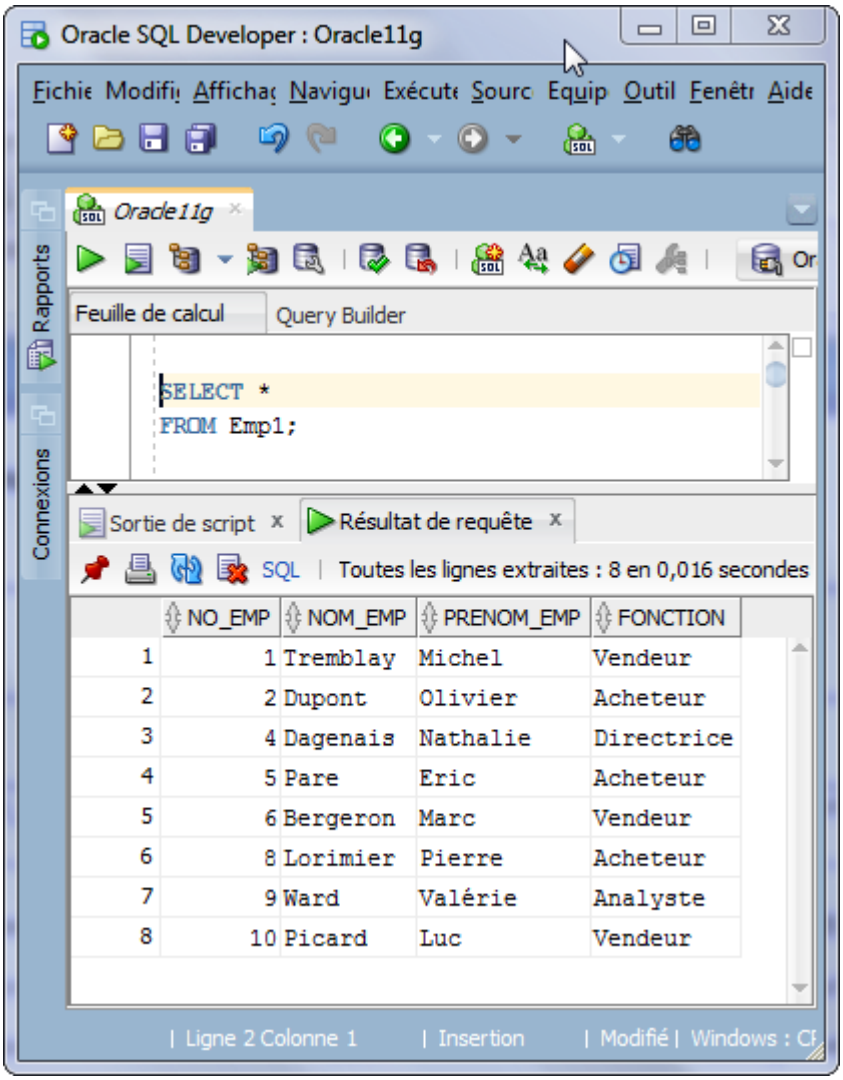
7.4 Modifier une vue

L'instruction **REPLACE VIEW** permet de modifier la définition de la vue.

EXEMPLE

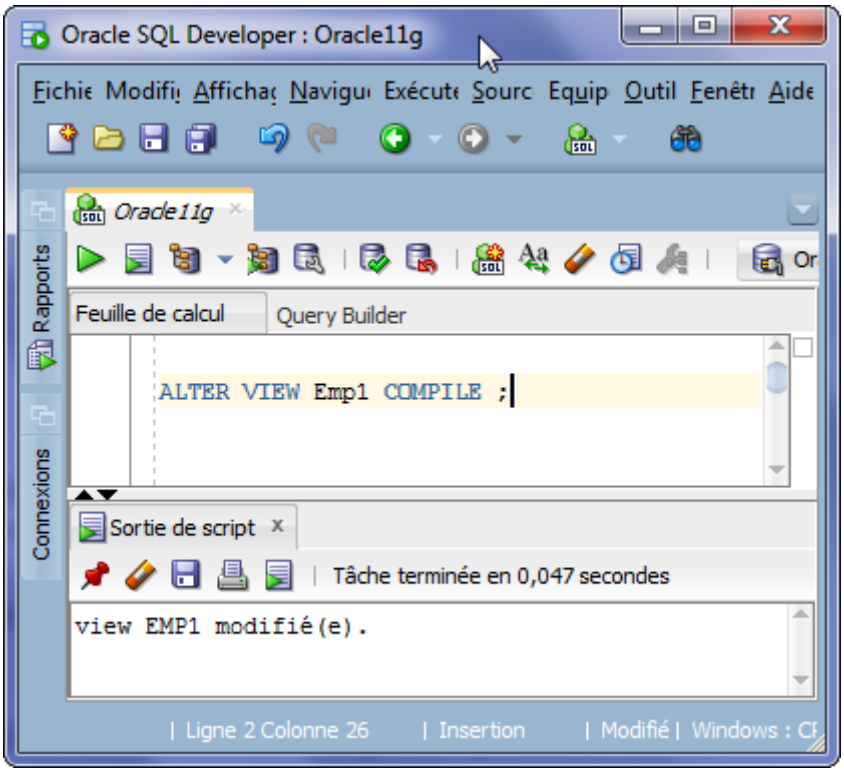


Utilisation de la vue :



7.5 **Recompiler une vue**

Il est possible de recompiler une vue avec la ALTER VIEW COMPILE :



8 GESTION DES TRANSACTIONS

8.1 Introduction

SQL garantit la cohérence des données par le biais des transactions. Les transactions vous offrent davantage de souplesse et un meilleur contrôle lors de la modification de données. Elles garantissent la cohérence des données en cas d'échec du processus utilisateur ou de panne du système.

Les transactions consistent en un ensemble d'ordres du LMD qui réalisent une modification cohérente des données. Par exemple, un transfert de fonds entre deux comptes implique de débiter un compte et d'en créditer un autre du même montant.

Les deux actions doivent, soit réussir, soit échouer en même temps : un crédit ne peut pas être validé sans le débit correspondant.

8.2 Quand Commence et Finit une Transaction ?

Une transaction commence dès le premier ordre SQL exécutable rencontré et se termine lorsque l'un des événements suivants se produit :

- Un ordre COMMIT ou ROLLBACK est lancé
- L'utilisateur quitte la session Base de données
- Il se produit une panne de machine ou du système d'exploitation

Lorsqu'une transaction prend fin, le prochain ordre SQL exécutable démarrera automatiquement la transaction suivante.

8.3 Ordres de Contrôle Explicite des Transactions

Vous avez la possibilité de contrôler la logique des transactions au moyen des ordres COMMIT, SAVEPOINT et ROLLBACK.

Ordre	**Description**
COMMIT	Met fin à la transaction courante en rendant définitives toutes les modifications de données en instance.
SAVE TRANSACTION [*nom*]	Pose une étiquette dans la transaction courante.
ROLLBACK	ROLLBACK met fin à la transaction courante et rejette toutes les modifications de données en instance. ROLLBACK TRANSACTION *name* annule toutes les

	modifications jusqu'au point de sauvegarde et supprime celui-ci.

8.4 Traitement Implicite des Transactions

État	Circonstances
Commit automatique	Exécution d'un ordre du LDD ou du LCD pour certains SGBD Sortie normale de l'analyseur de requêtes, sans précision d'un ordre COMMIT ou ROLLBACK explicite
Rollback automatique	Fin anormale de l'analyseur de requêtes ou panne du système

8.5 État des Données Avant COMMIT ou ROLLBACK

- Les opérations de manipulation des données se déroulant principalement dans le buffer de la base de données, il est possible de restaurer l'état précédent des données.
- L'utilisateur courant peut afficher le résultat de ses opérations de manipulation de données en interrogeant les tables.
- Les autres utilisateurs ne peuvent pas voir le résultat des opérations de manipulation de données réalisées par l'utilisateur courant. SQL met en œuvre un principe de lecture cohérente qui garantit que l'utilisateur voit les données telles qu'elles se présentaient lors de la dernière validation.
- Les lignes concernées par la transaction sont verrouillées ; aucun autre utilisateur ne peut modifier les données qu'elles contiennent.

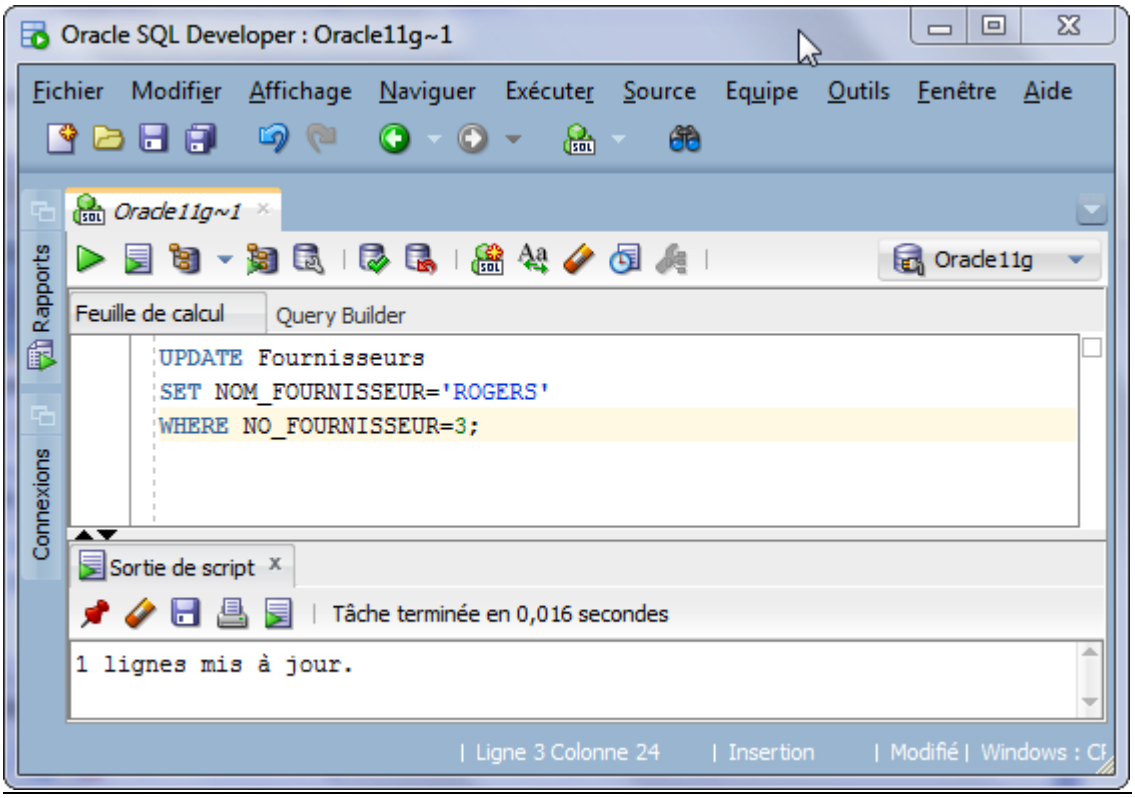
8.6 État des Données APRES COMMIT

Pour enregistrer définitivement les modifications en instance, utilisez l'ordre COMMIT. Après l'exécution d'un ordre COMMIT :

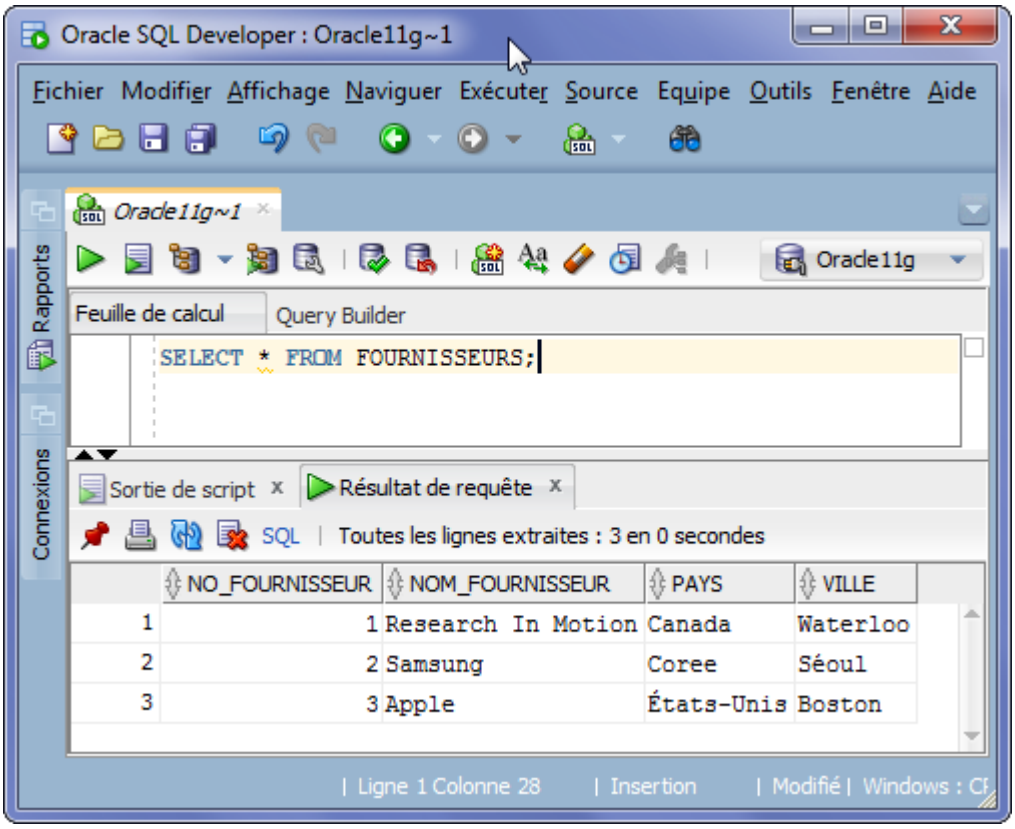
- Les modifications de données sont écrites définitivement dans la base de données.
- L'état précédent des données est irrémédiablement perdu.
- Tous les utilisateurs peuvent voir les résultats de la transaction.
- Les lignes qui étaient verrouillées sont libérées et redeviennent accessibles à d'autres utilisateurs pour modification.
- Tous les points de sauvegarde sont effacés.

EXEMPLE

Faire la transaction de mise à jour suivante :



Sans quitter la session courante et à partir d'une nouvelle session faire l'interrogation suivante :



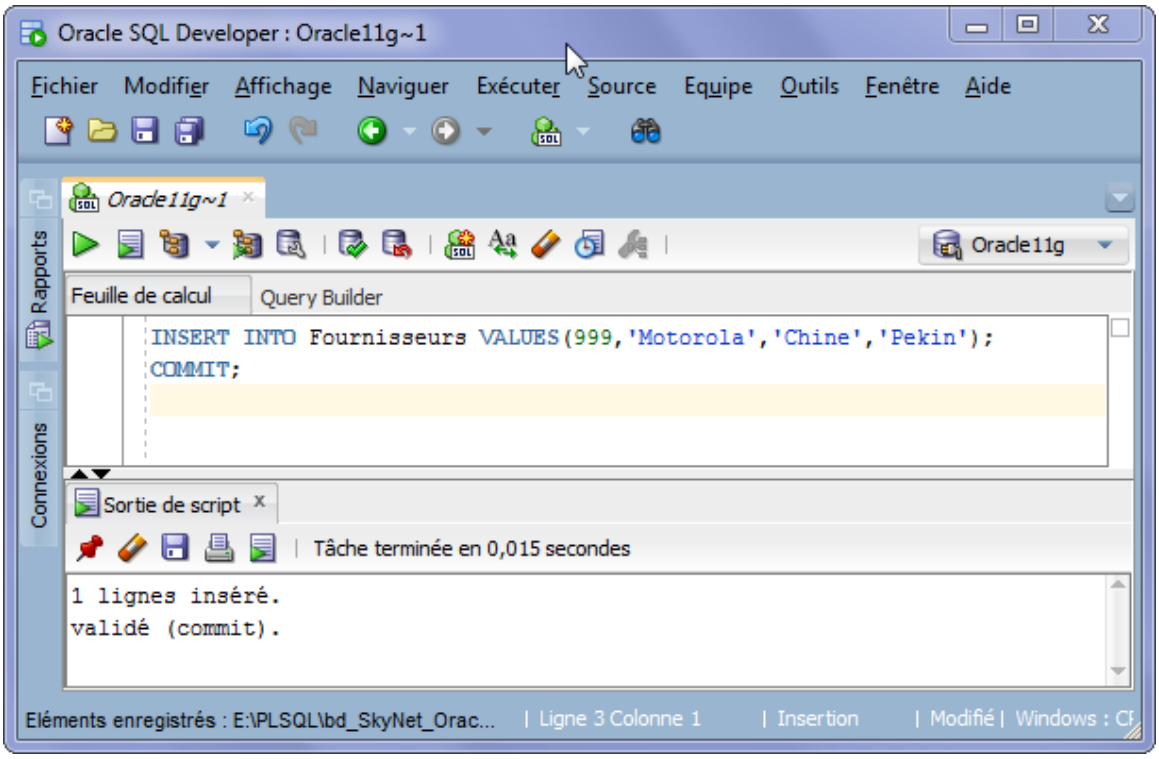
CONCLUSION

Les transactions non validées ne sont pas visibles dans les autres sessions tant qu'elles ne sont pas validées.

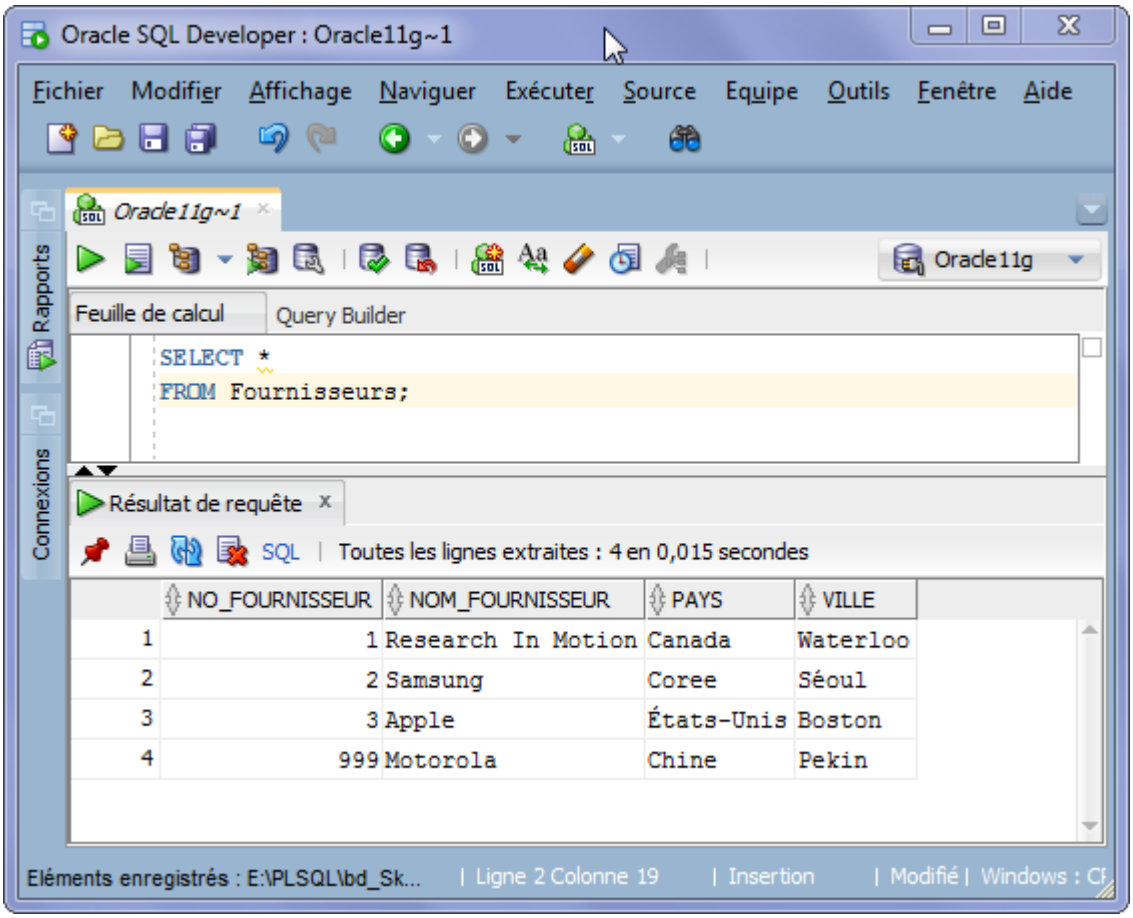
EXEMPLE AVEC COMMIT

Ajouter un nouveau fournisseur dans la table FOURNISSEURS.

Ensuite valider la transaction.



Ouvrir une nouvelle session et vérifier que la transaction a bien été validée :

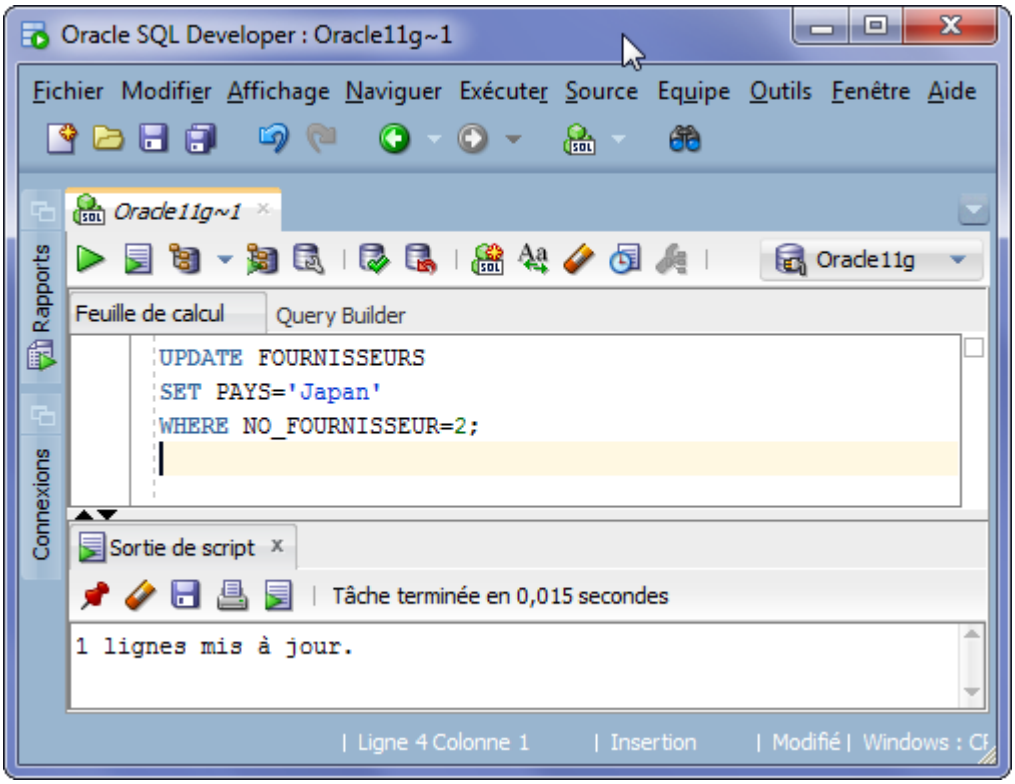


CONCLUSION

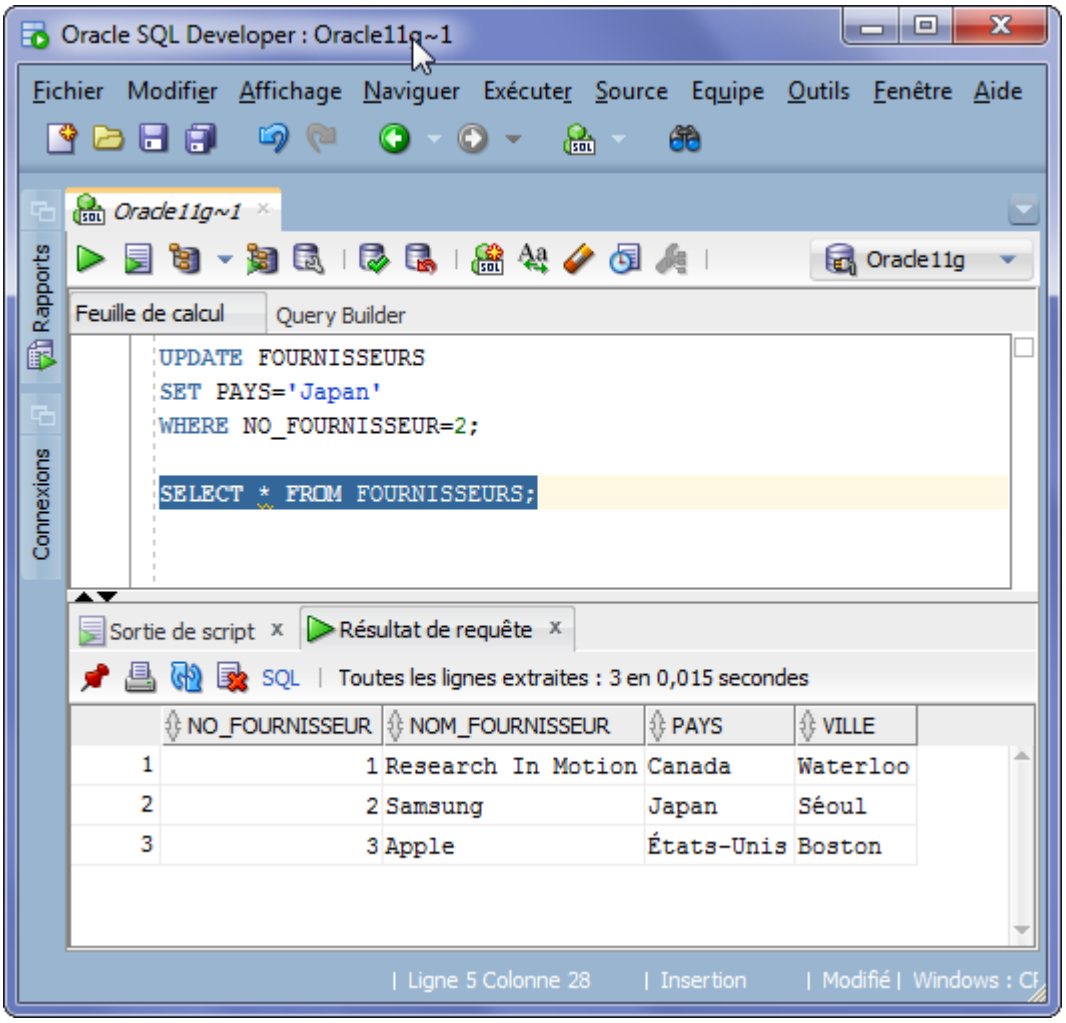
Les transactions validées sont visibles dans les autres sessions.

EXEMPLE AVEC ROLLBACK

Faire la transaction suivante :

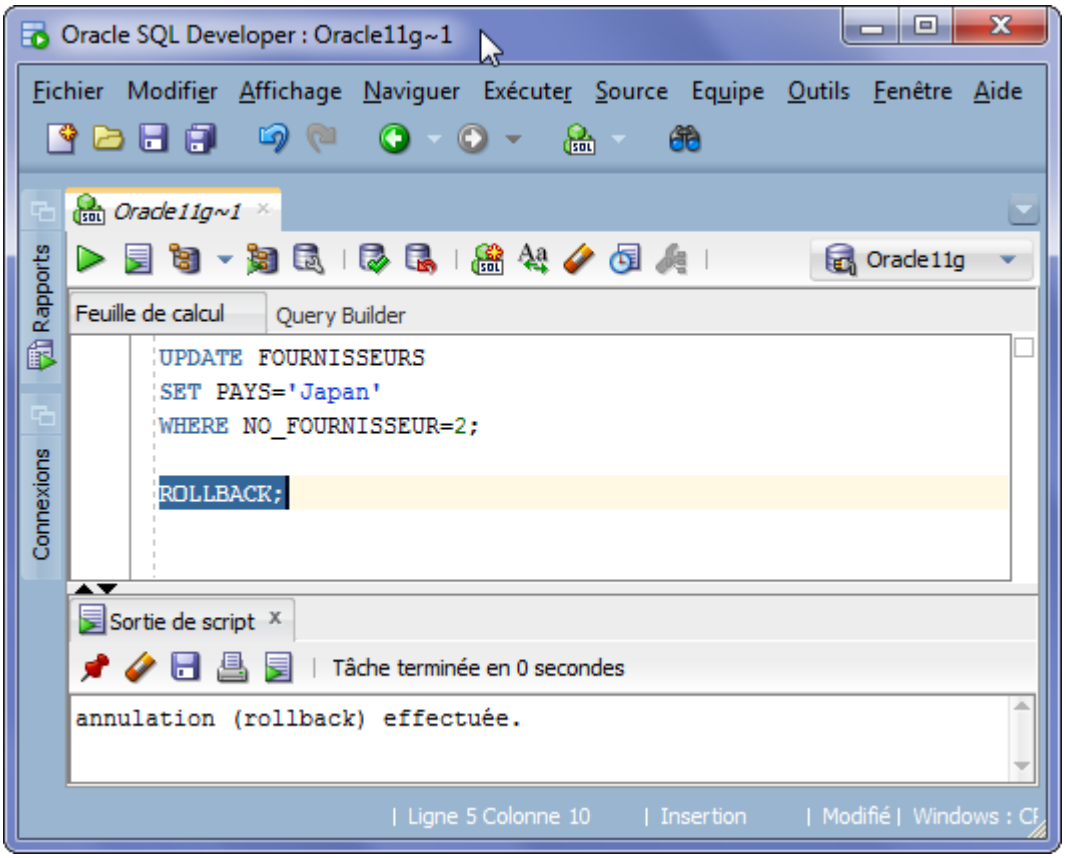


Interroger les données dans la même session :

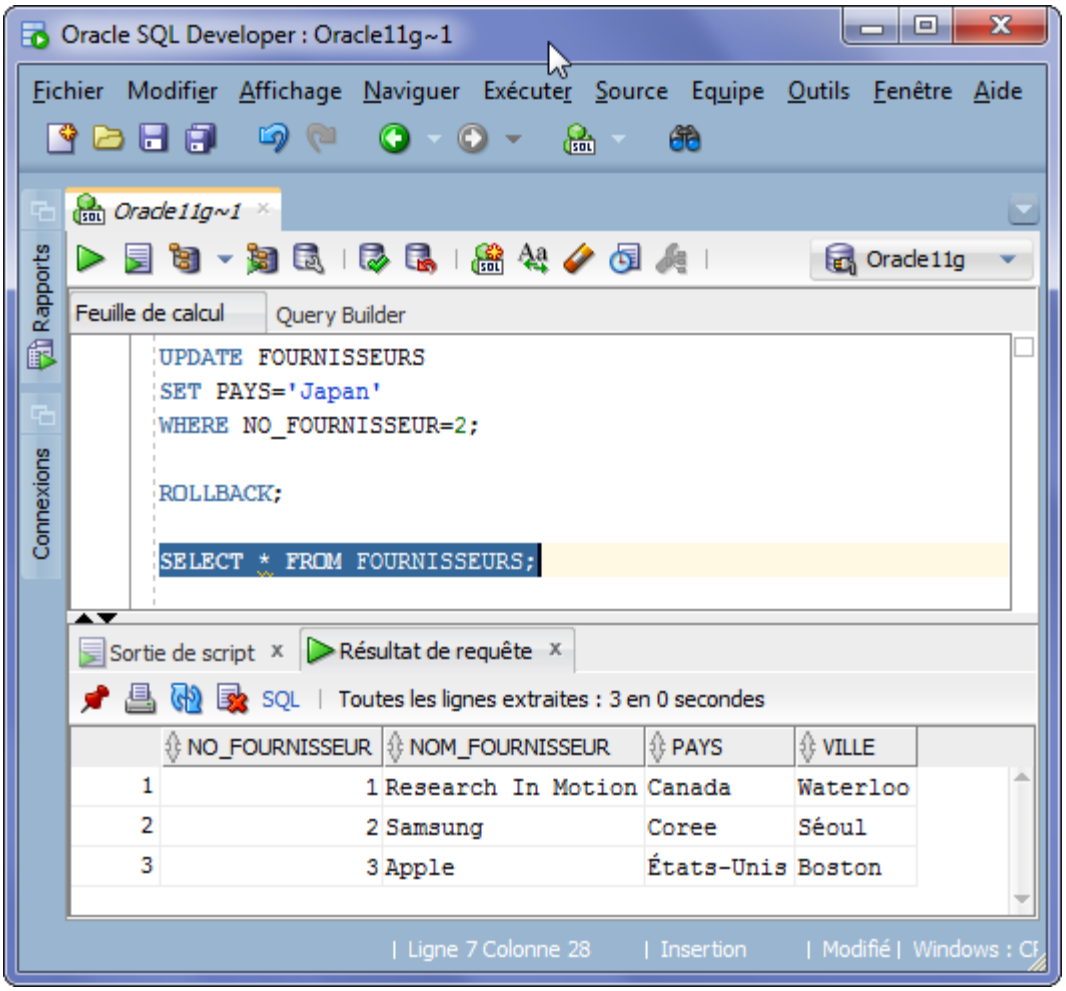


On voit bien que la transaction est visible même si elle n'a pas encore été validée.

Annuler la transaction :



Interroger les données de nouveau :



La transaction a bien été annulée.

ROLLBACK AVEC SAVEPOINT

Il est possible d'annuler une transaction et de retourner à un point spécifique en utilisant la commande : `ROLLBACK [To] NOM_SAVEPOINT;`

```
BEGIN
  SAVEPOINT point1;
  Transaction 1;
  SAVEPOINT point2;
  Transaction 2;
EXCEPTION
WHEN OTHERS THEN
      ROLLBACK TO SAVEPOINT point2;
END;
```

9 LES TRIGGERS

Les déclencheurs ou **triggers** servent à étendre les mécanismes de gestion de l'intégrité à des contraintes complexes et permettre le contrôle de saisie. Il s'agit de code déclenché lors de certains événements de la base de données. Un déclencheur est toujours rattaché à une table ou à une vue.

Il s'agit de code déclenché lors de certains événements de la base de données. Un trigger est toujours rattaché à une table.

Les événements qui déclenchent un trigger sont :

- L'insertion de données (INSERT)
- La suppression de données (DELETE)
- La mise à jour (UPDATE)

Ils sont donc déclenchés systématiquement par une instruction SQL, après l'exécution de cette requête (**AFTER**), ou à la place de l'exécution de cette requête (**INSTEAD**).

SQL n'implémente pas de trigger BEFORE (avant exécution), mais nous verrons comment le simuler...

Les pseudos tables **INSERTED** et **DELETED** contiennent les données respectives de l'insertion ou la mise à jour (**INSERTED**) ou bien de la suppression (**DELETED**). On peut les utiliser dans des requêtes comme des tables ordinaires.

La structure de ces tables est calquée sur la structure de la table ou vue sur laquelle repose le trigger.

9.1 Creation d'un trigger

La syntaxe de base d'un déclencheur est la suivante :

```
CREATE OR REPLACE TRIGGER <nom_trigger>
  AFTER|BEFORE INSERT or UPDATE or DELETE
  ON <nom_table>
  FOR EACH ROW

DECLARE
  <Section déclaration des variables>
BEGIN
    <Code de la logique du trigger>
EXCEPTION
    <Gestion des exceptions>
END < nom_trigger >
/
```

AFTER

Indique que le déclencheur est déclenché uniquement lorsque toutes les opérations spécifiées dans l'instruction SQL de déclenchement ont été exécutées avec succès.

Toutes les actions d'intégrité référentielle en cascade et les vérifications de contrainte doivent aussi réussir pour que ce déclencheur puisse s'exécuter.

AFTER est la valeur par défaut, si FOR est le seul mot-clé spécifié.

Les déclencheurs AFTER ne peuvent pas être définis sur des vues.

{ [DELETE] [,] [INSERT] [,] [UPDATE] }

Mots clés qui spécifient les instructions de modification des données qui, lorsqu'elles sont exécutées sur cette table ou vue, activent le déclencheur. Vous devez spécifier au moins une option. Vous pouvez combiner toutes les options dans n'importe quel ordre dans la définition du déclencheur. Si plusieurs options sont spécifiées, séparez-les par des virgules.

Emplois typiques :

- suppression, insertion et mise à jour en cascade
- contrôle de validité
- respect d'intégrité complexe
- formatage de données
- archivage automatique
- ...

9.2 **Fonction UPDATING**

La fonction UPDATING permet de tester si une colonne est visée par un changement de valeur.

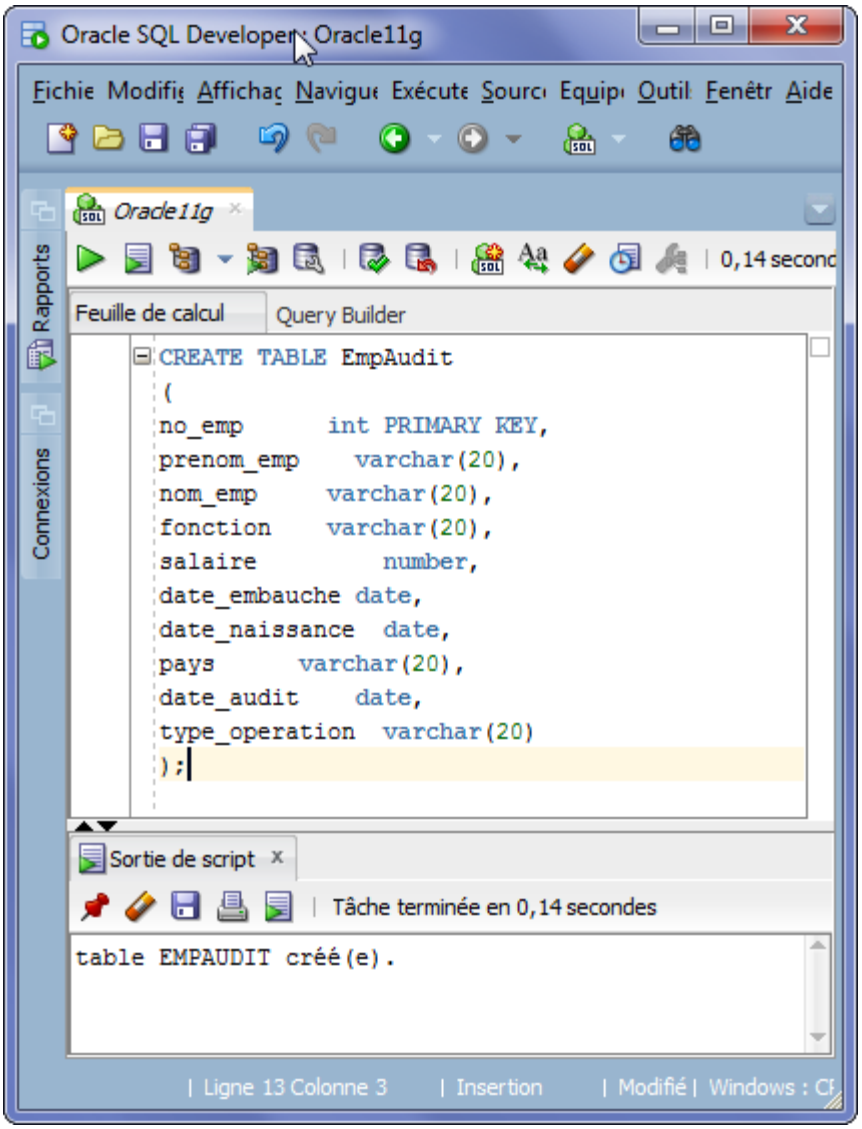
Elle s'emploie de la manière suivante :

```
IF [NOT] UPDATING(<colonne>) THEN
   <traitement>
END IF
```

Table à créer avant de faire les exemples :

On va créer une nouvelle table **EMPAudit** pour stocker les informations d'audit sur la table **Employes**.

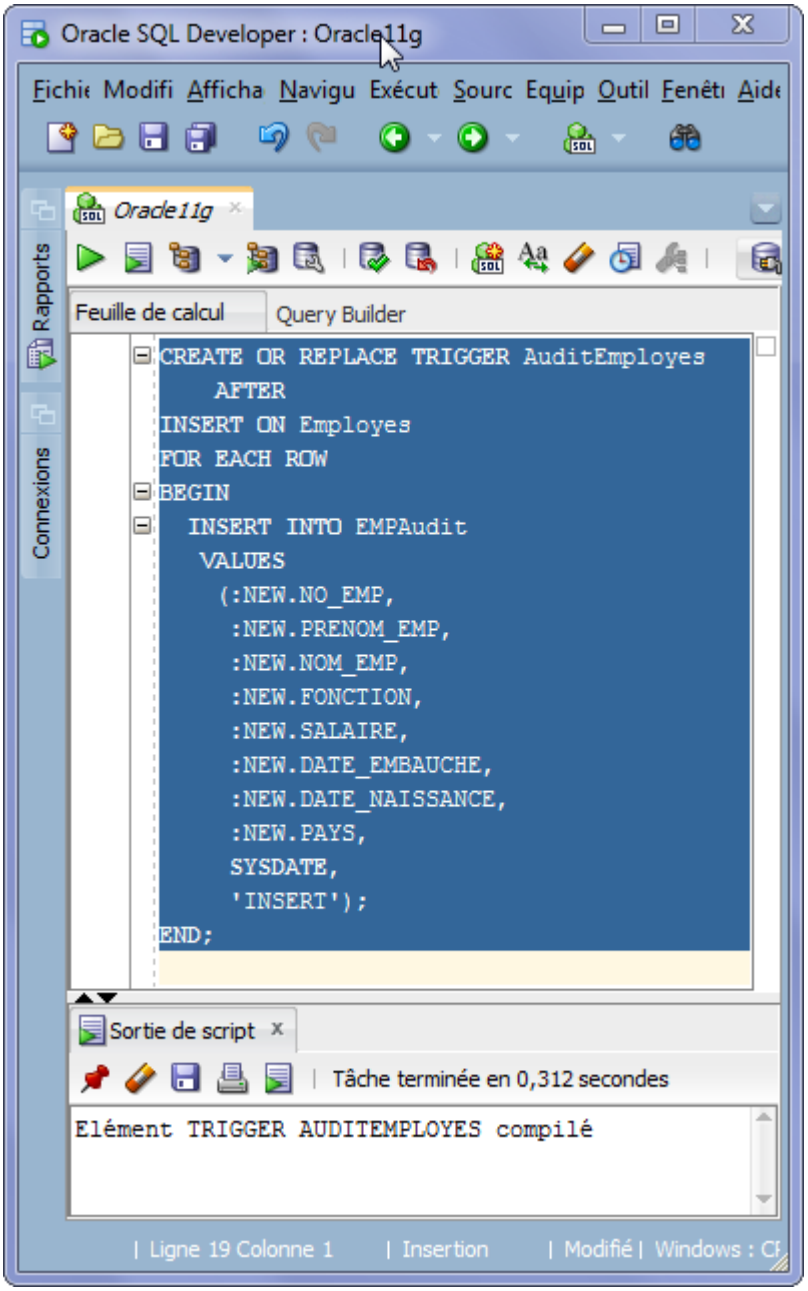
```
CREATE TABLE EmpAudit
(
no_emp          int PRIMARY KEY,
prenom_emp      varchar(20),
nom_emp         varchar(20),
fonction        varchar(20),
salaire         number,
date_embauche   date,
date_naissance  date,
pays            varchar(20),
date_audit      date,
type_operation  varchar(20)
);
```


Oracle SQL Developer : Oracle11g
Fichie Modifi Affichag Navigue Exécute Sourc Equip Outil Fenêtr Aide
Oracle11g
Rapports
Connexions
Feuille de calcul
Query Builder
CREATE TABLE EmpAudit
(
no_emp int PRIMARY KEY,
prenom_emp varchar(20),
nom_emp varchar(20),
fonction varchar(20),
salaire number,
date_embauche date,
date_naissance date,
pays varchar(20),
date_audit date,
type_operation varchar(20)
);
Sortie de script
Tâche terminée en 0,14 secondes
table EMPAUDIT créé(e).
Ligne 13 Colonne 3
Insertion
Modifié
Windows : Cr

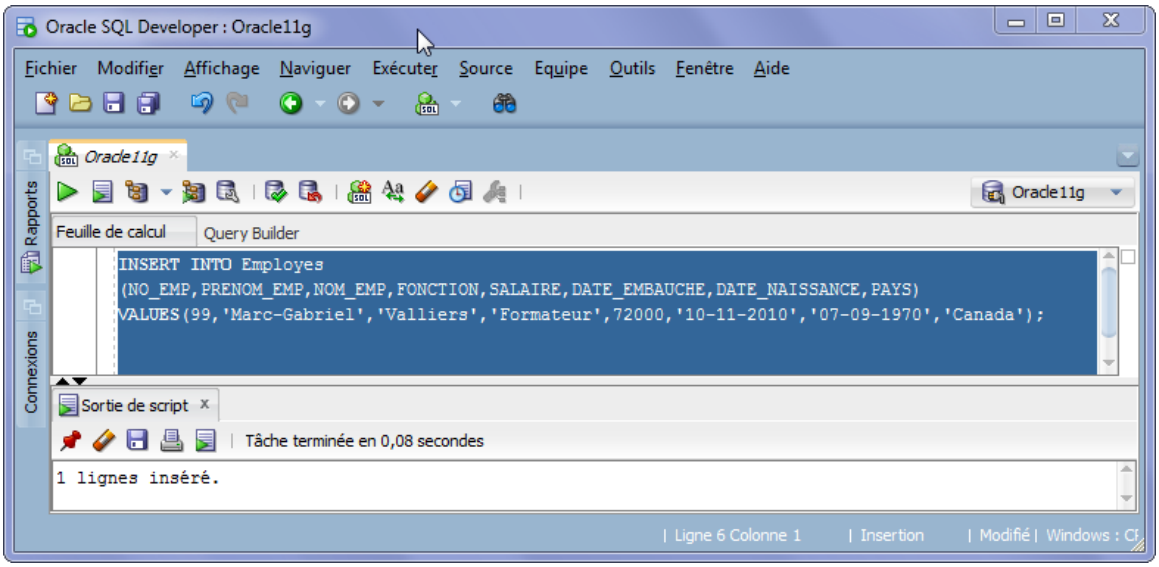
9.3 Trigger AFTER INSERT

EXEMPLE

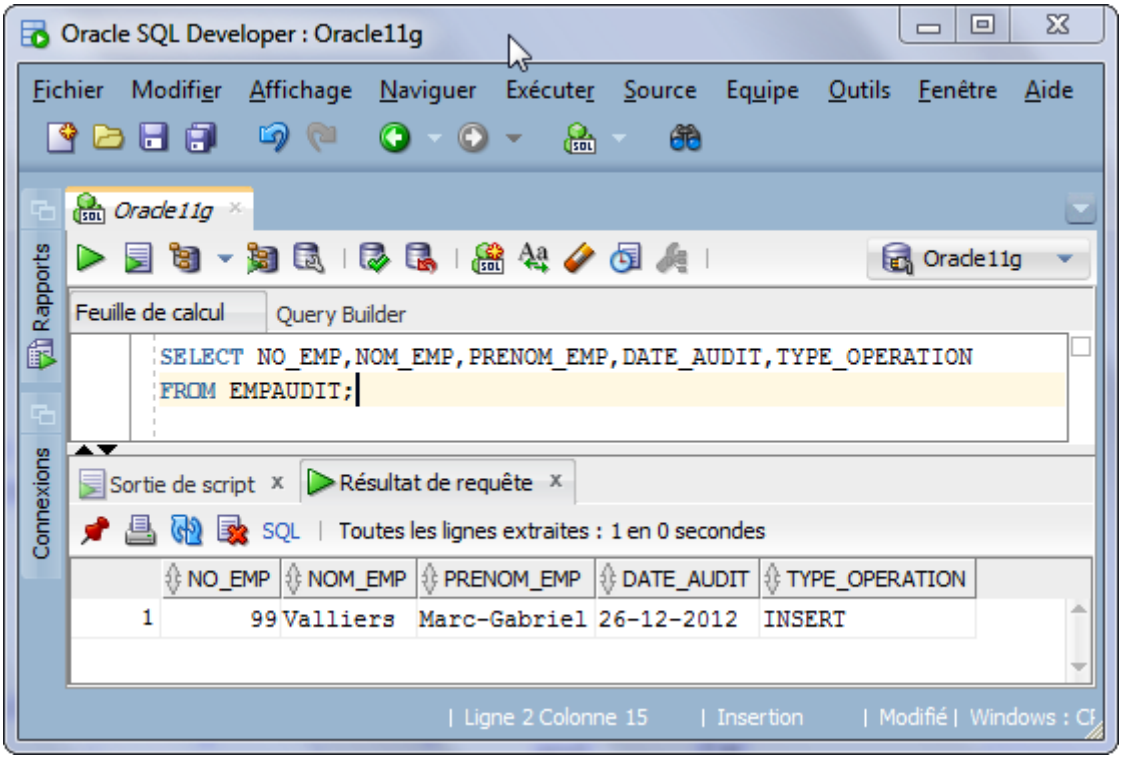
On va créer un trigger qui va se déclencher après chaque ligne insérer dans la table **Employes**. Le trigger va insérer la même ligne dans la table **EMPAudit**.



Pour que le trigger se déclenche, on va insérer un nouvel employé dans la table **EMPLOYES**.



Vérifier la table d'audit :

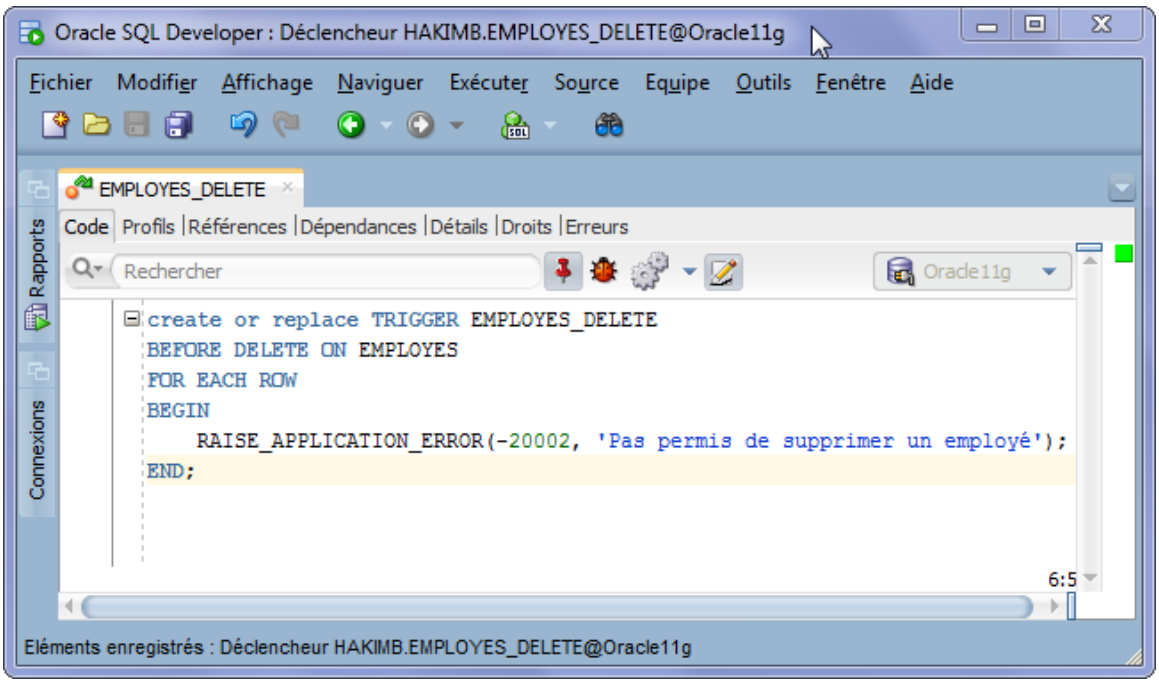


L'insert a bien été audité dans la table **EMPAUDIT**.

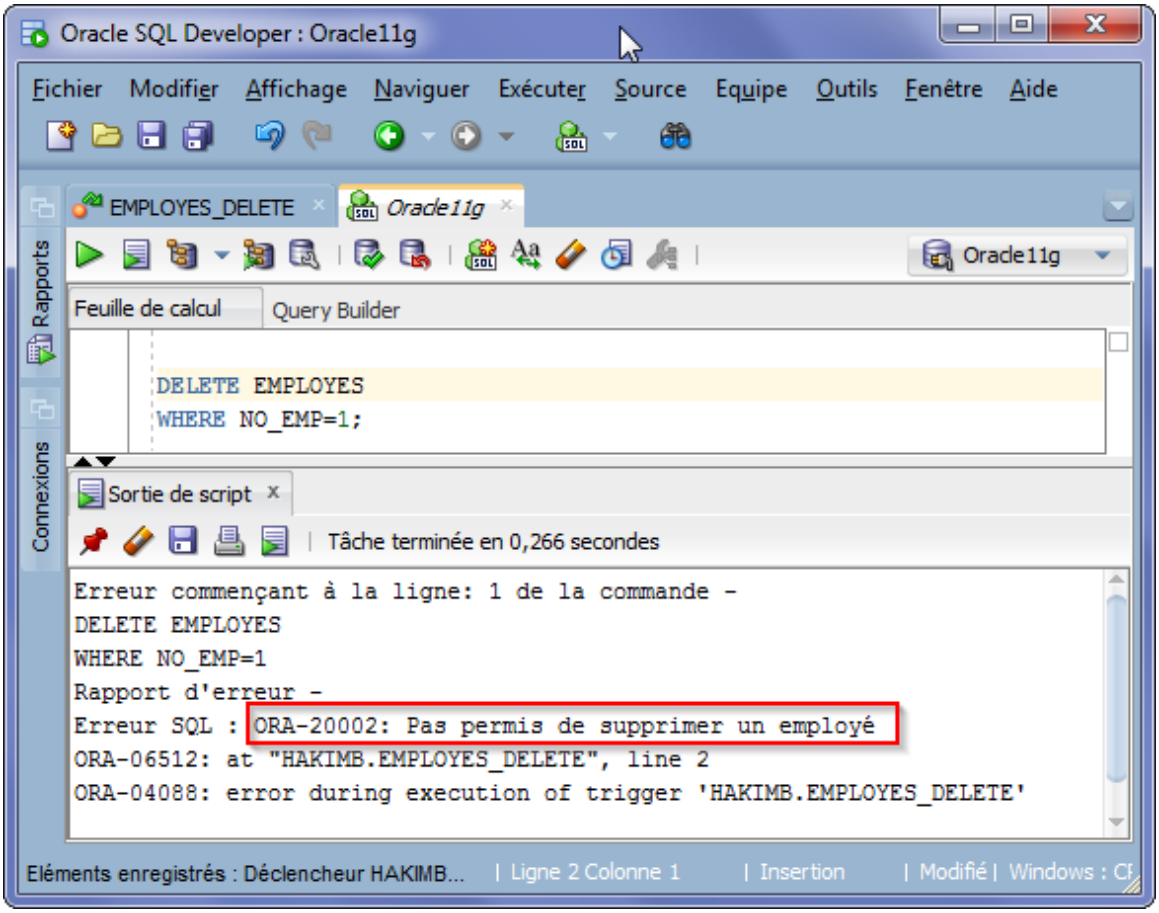
9.4 Trigger BEFORE DELETE

EXEMPLE

Créer un trigger qui empêche la suppression d'un employé de la table EMPLOYES:



Tester le trigger :

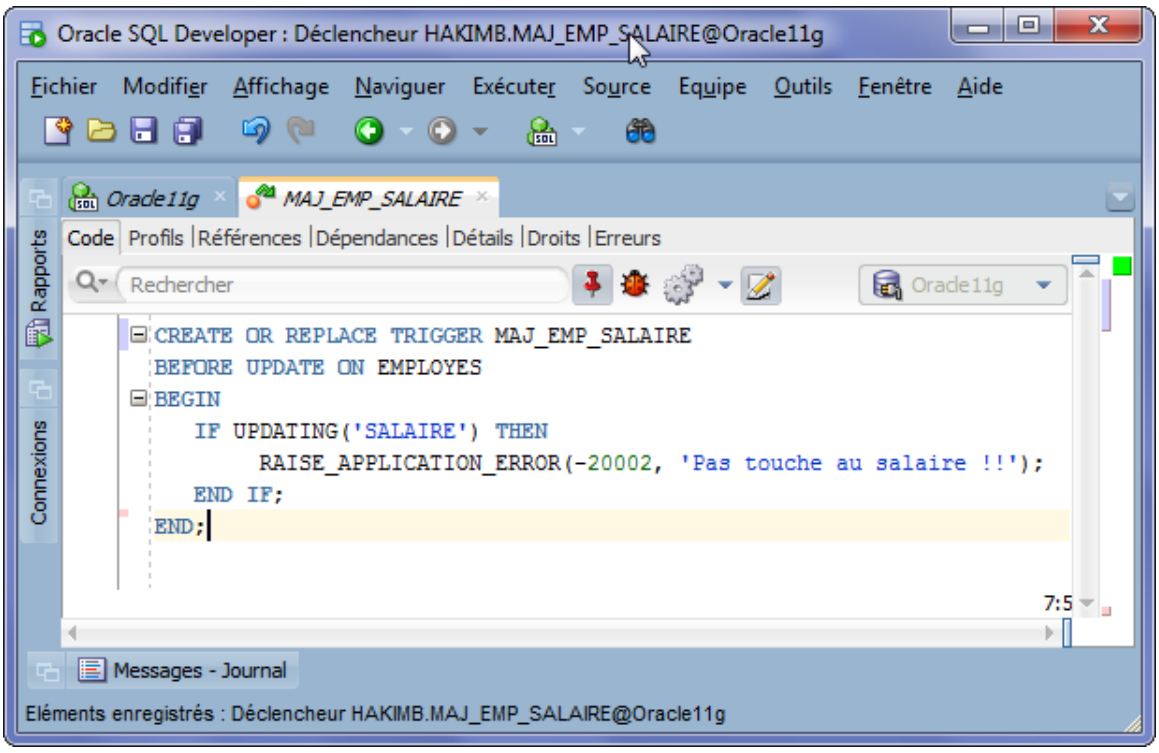


Comme on peut le voir, il est impossible de supprimer un employé.

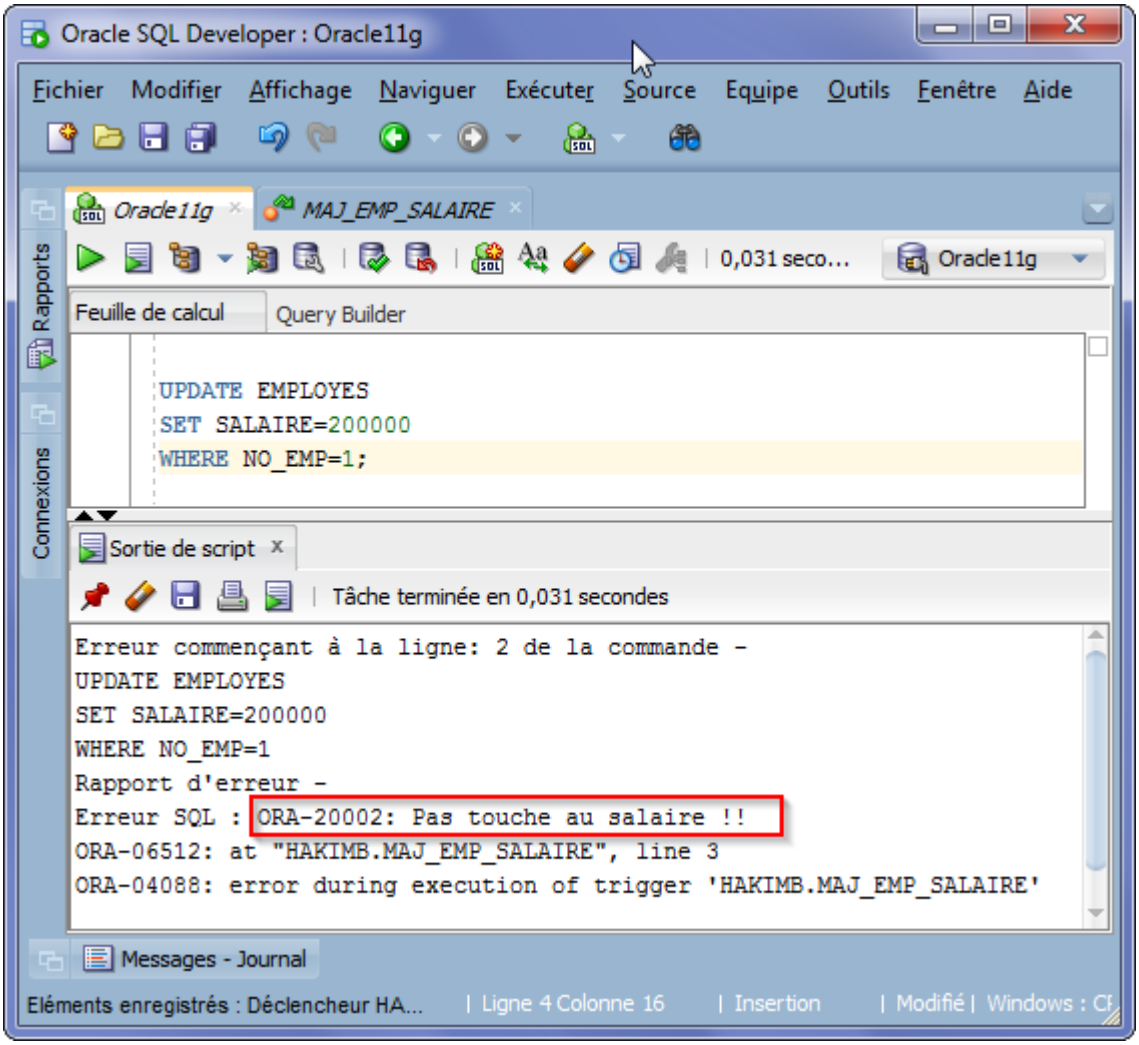
9.5 Trigger BEFORE UPDATE

EXEMPLE

Interdire la mise à jour du salaire sur la table Employes :



Tester le trigger :



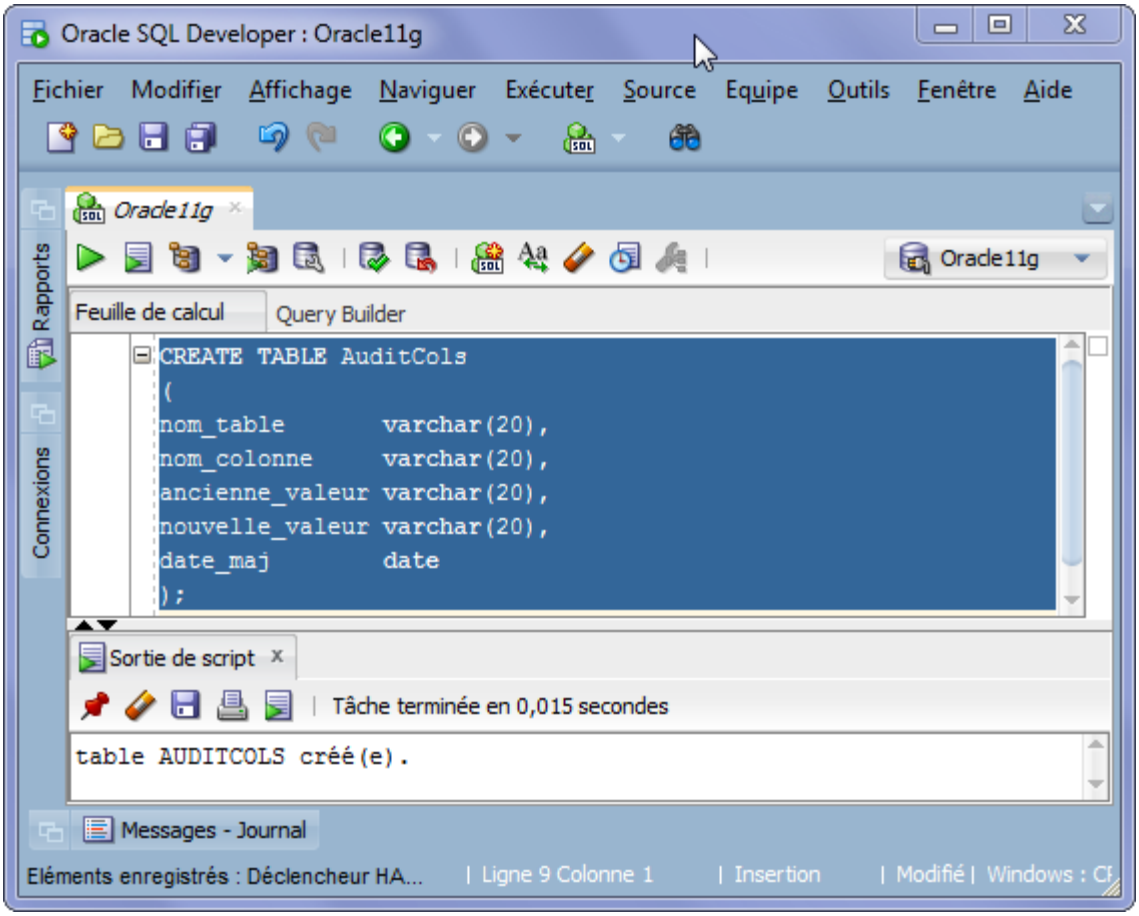
Comme on peut le voir, il est impossible de modifier le salaire !!

9.6 Auditer les modifications sur des colonnes spécifiques

Dans certain cas on aimerait auditer les modifications sur des colonnes spécifiques telles que l'adresse et le numéro de téléphone en concevant l'ancienne et la nouvelle valeur ainsi que la date du changement.

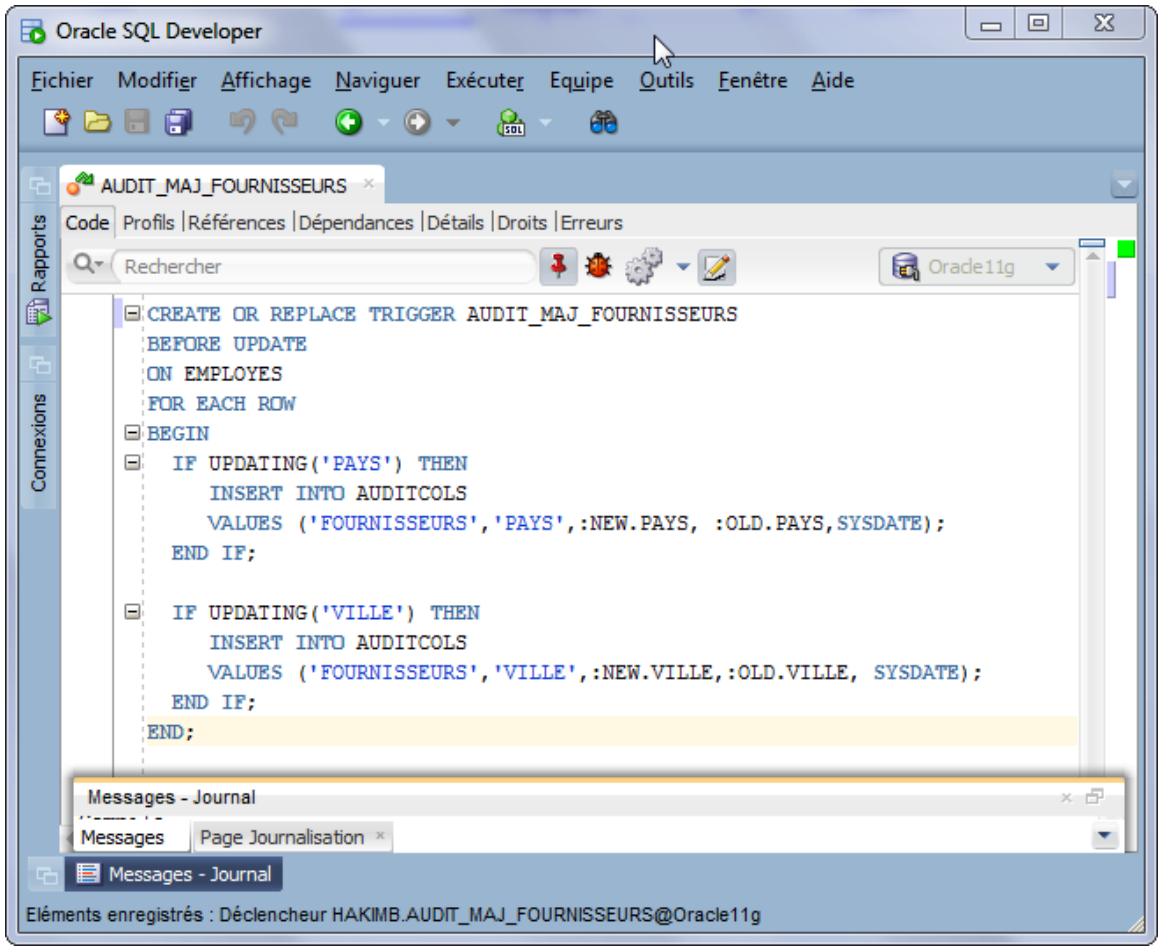
Dans ce cas, on va avoir besoin d'une table d'audit pour conserver ces valeurs :

```
CREATE TABLE AuditCols
(
nom_table        varchar(20),
nom_colonne      varchar(20),
ancienne_valeur  varchar(20),
nouvelle_valeur  varchar(20),
date_maj         date
);
```



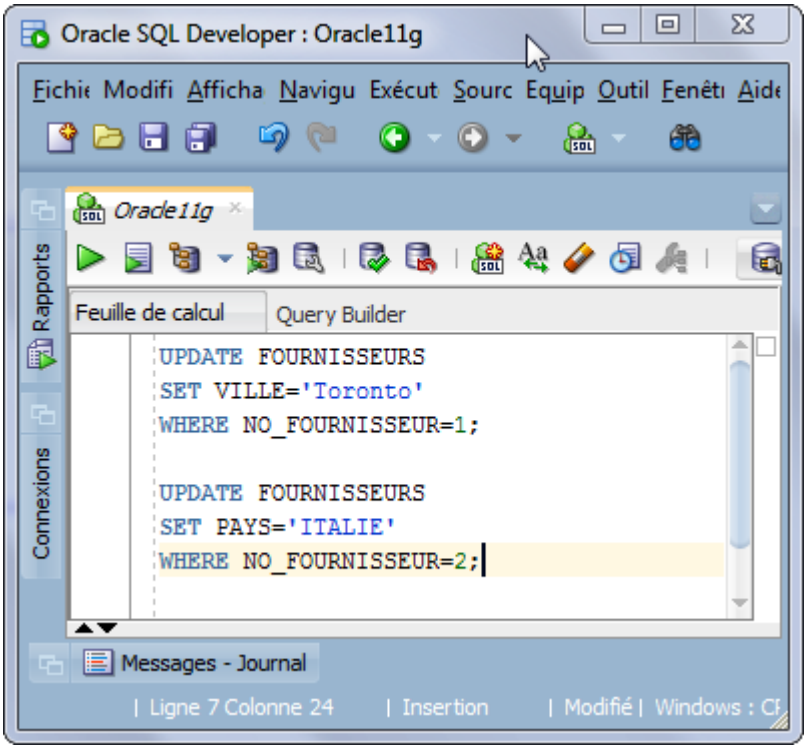
Voici la définition du trigger :

```
CREATE OR REPLACE TRIGGER AUDIT_MAJ_FOURNISSEURS
BEFORE UPDATE
ON EMPLOYES
FOR EACH ROW
BEGIN
  IF UPDATING('PAYS') THEN
     INSERT INTO AUDITCOLS
 VALUES ('FOURNISSEURS','PAYS',:NEW.PAYS, :OLD.PAYS,SYSDATE);
  END IF;

  IF UPDATING('VILLE') THEN
     INSERT INTO AUDITCOLS
     VALUES     ('FOURNISSEURS','VILLE',:NEW.VILLE,:OLD.VILLE,
SYSDATE);
  END IF;
END;
```



Tester le trigger :

```
UPDATE FOURNISSEURS
SET VILLE='Toronto'
WHERE NO_FOURNISSEUR=1;

UPDATE FOURNISSEURS
SET PAYS='ITALIE'
WHERE NO_FOURNISSEUR=2;
```




```
SELECT * FROM AUDITCOLS;
```

9.7 Trigger de base de données

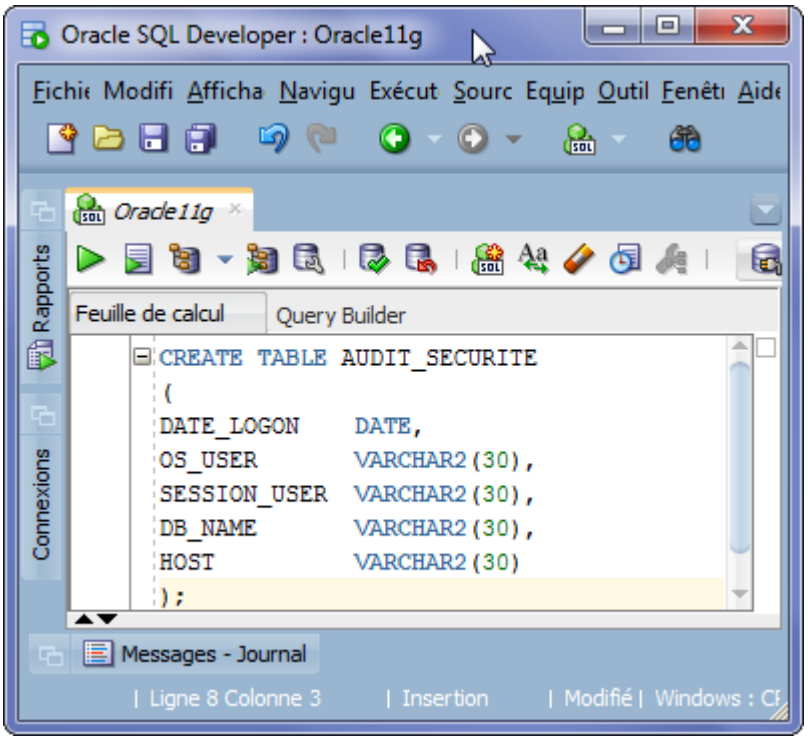
On se propose de créer un trigger qui permet de journaliser les connexions à la base de données.

Les informations journalisées sont les suivantes :

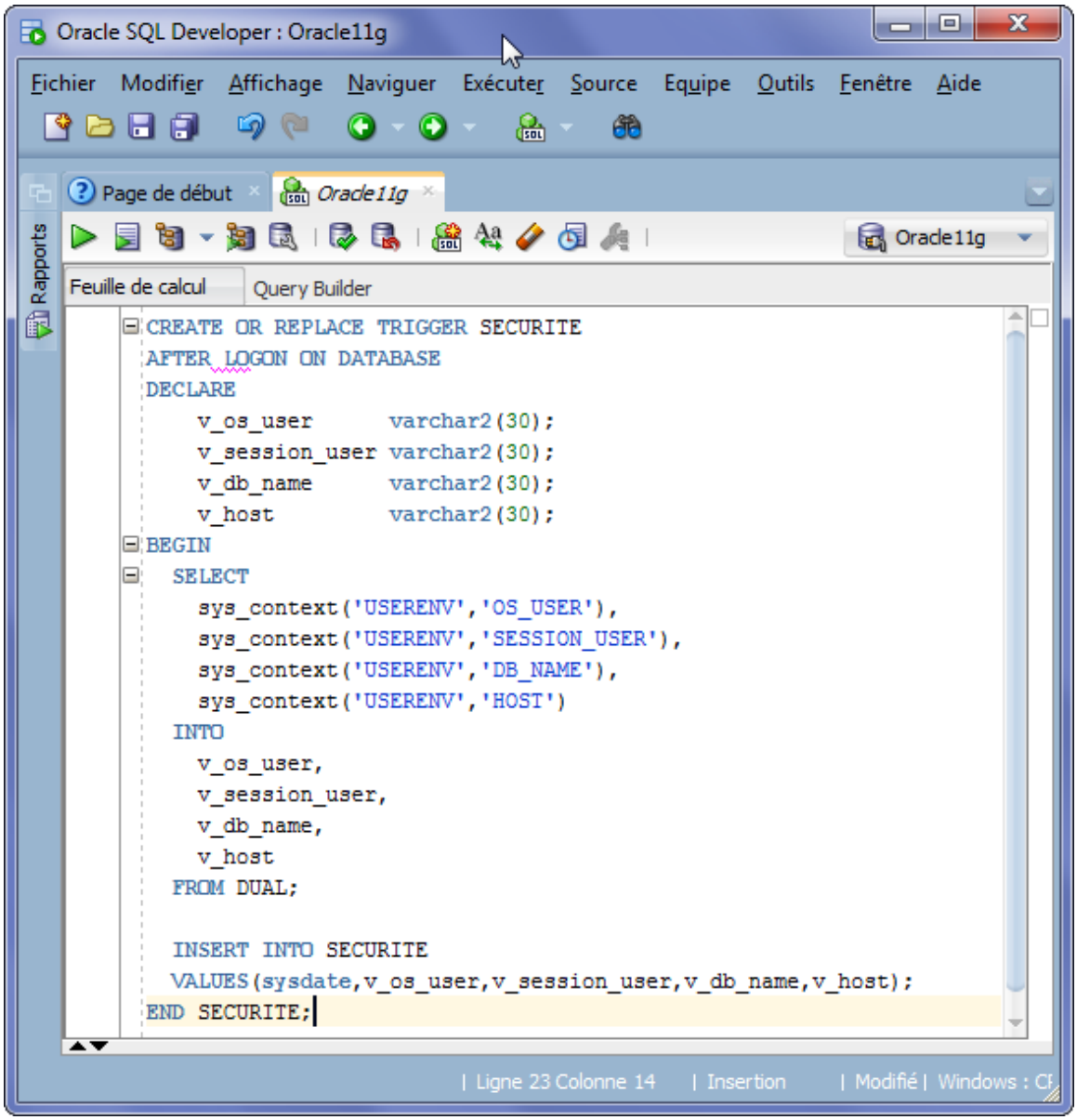
- Date de connexion
- Usager du système d'exploitation (OS_USER)
- Usager de la base de données (**SESSION_USER**)
- Nom de la base de données (DB_NAME)
- Serveur (HOST)

La table AUDIT_SECURITE va servir à stocker ces informations :

```
CREATE TABLE AUDIT_SECURITE
(
DATE_LOGON      DATE,
OS_USER         VARCHAR2(30),
SESSION_USER    VARCHAR2(30),
DB_NAME         VARCHAR2(30),
HOST            VARCHAR2(30)
);
```

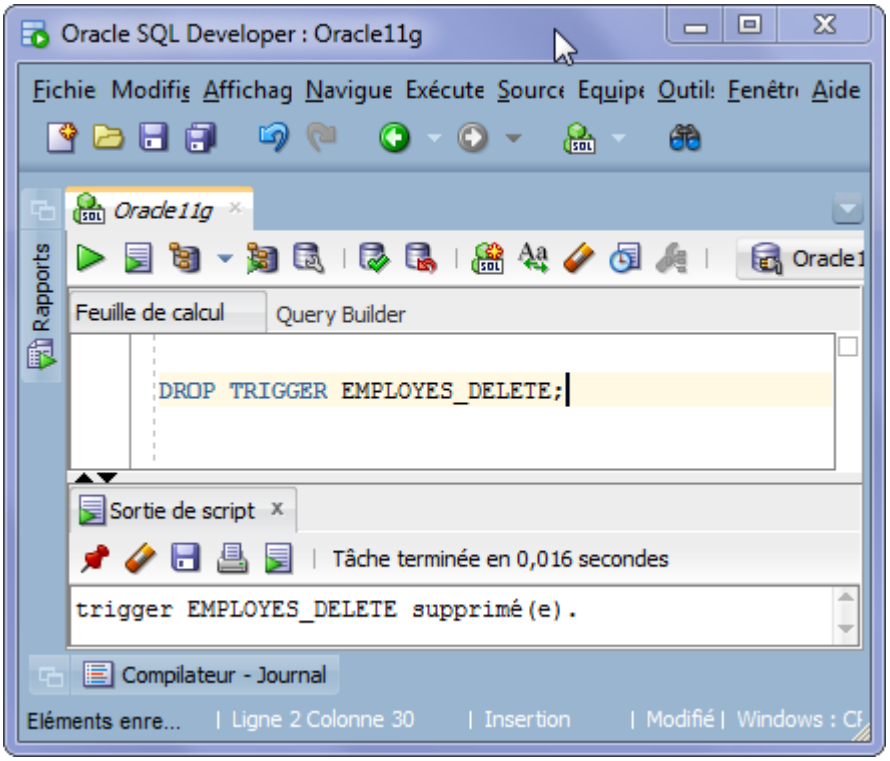


```
CREATE OR REPLACE TRIGGER SECURITE
AFTER LOGON ON DATABASE
DECLARE
    v_os_user      varchar2(30);
    v_session_user varchar2(30);
    v_db_name      varchar2(30);
    v_host         varchar2(30);
BEGIN
  SELECT
    sys_context('USERENV','OS_USER'),
    sys_context('USERENV','SESSION_USER'),
    sys_context('USERENV','DB_NAME'),
    sys_context('USERENV','HOST')
  INTO
    v_os_user,
    v_session_user,
    v_db_name,
    v_host
  FROM DUAL;

  INSERT INTO SECURITE
  VALUES(sysdate,v_os_user,v_session_user,v_db_name,v_host);
END SECURITE;
```

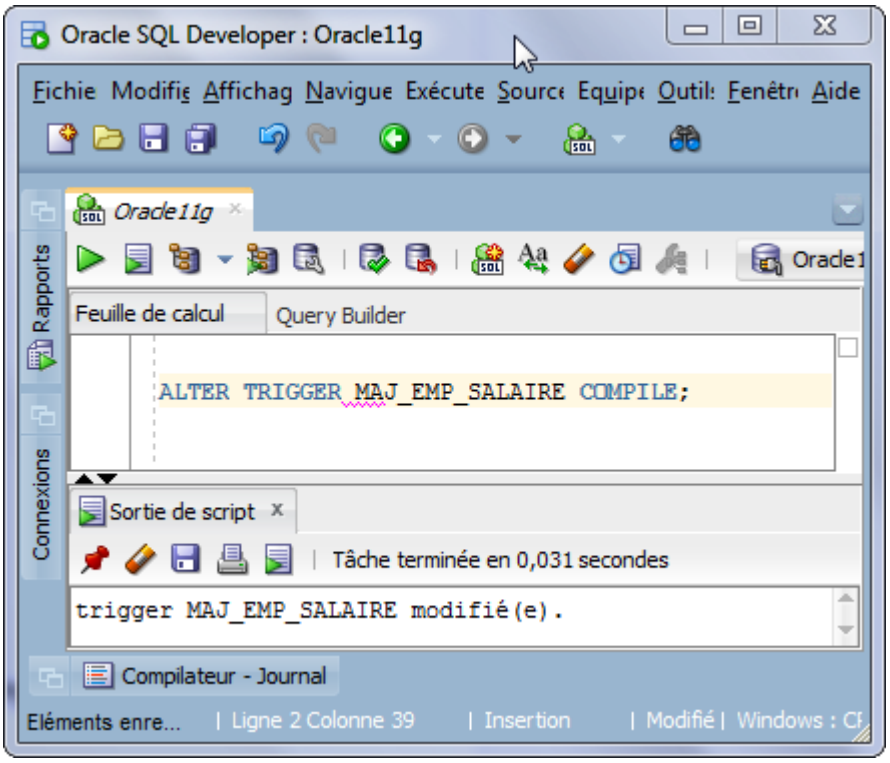

Oracle SQL Developer : Oracle11g
Fichier Modifier Affichage Naviguer Exécuter Source Equipe Outils Fenêtre Aide
Page de début
Oracle11g
Rapports
Oracle11g
Feuille de calcul
Query Builder
CREATE OR REPLACE TRIGGER SECURITE
AFTER LOGON ON DATABASE
DECLARE
v_os_user varchar2(30);
v_session_user varchar2(30);
v_db_name varchar2(30);
v_host varchar2(30);
BEGIN
SELECT
sys_context('USERENV','OS_USER'),
sys_context('USERENV','SESSION_USER'),
sys_context('USERENV','DB_NAME'),
sys_context('USERENV','HOST')
INTO
v_os_user,
v_session_user,
v_db_name,
v_host
FROM DUAL;
INSERT INTO SECURITE
VALUES(sysdate,v_os_user,v_session_user,v_db_name,v_host);
END SECURITE;
Ligne 23 Colonne 14
Insertion
Modifié
Windows : CF

9.8 Supprimer un trigger

Supprimer le trigger **`Employes_Delete.`**



9.9 Recompiler un trigger



9.10 Modifier la définition d'un trigger

```
CREATE OR REPLACE TRIGGER Employes_Delete

...
```

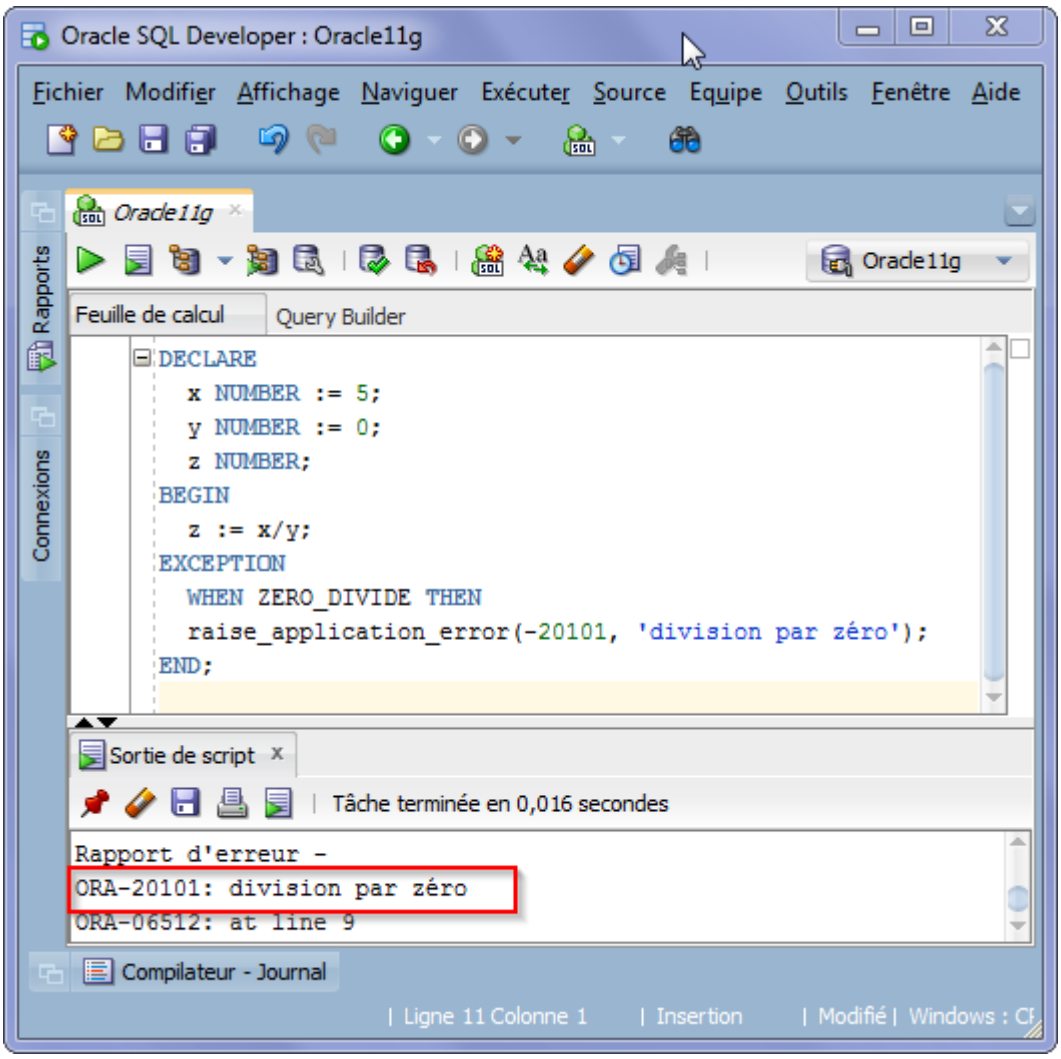
10 GESTION DES ERREURS

10.1 Traitement des execptions

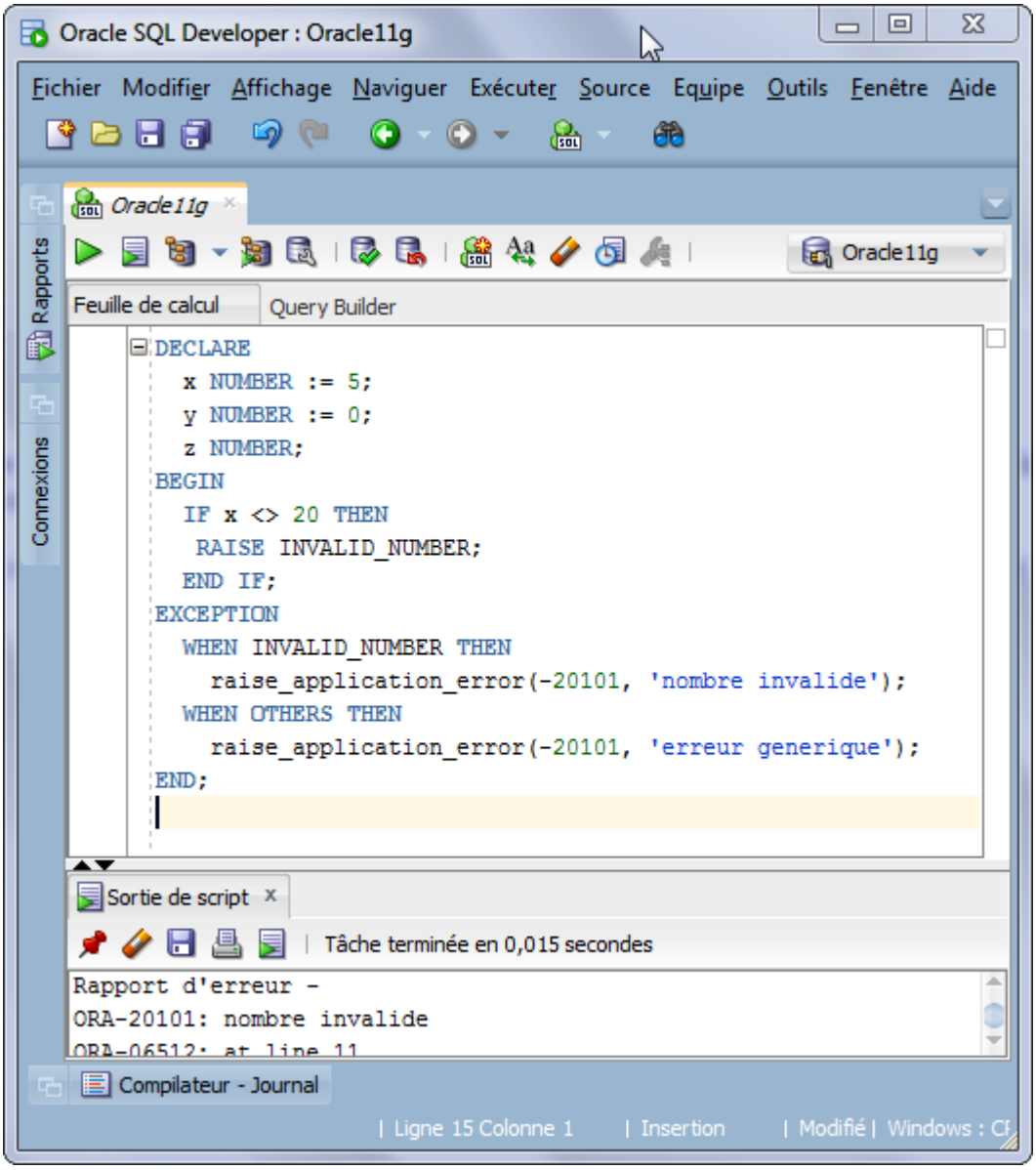
Le traitement des exceptions se fait dans la section EXCEPTION.

Exception ZERO_DIVIDE

```
DECLARE
  x NUMBER := 5;
  y NUMBER := 0;
  z NUMBER;
BEGIN
  z := x/y;
EXCEPTION
  WHEN ZERO_DIVIDE THEN
  raise_application_error(-20101, 'division par zéro');
END;
```


Oracle SQL Developer : Oracle11g
Fichier Modifier Affichage Naviguer Exécuter Source Equipe Outils Fenêtre Aide
Oracle11g
Rapports
Connexions
Oracle11g
Feuille de calcul
Query Builder
DECLARE
x NUMBER := 5;
y NUMBER := 0;
z NUMBER;
BEGIN
z := x/y;
EXCEPTION
WHEN ZERO_DIVIDE THEN
raise_application_error(-20101, 'division par zéro');
END;
Sortie de script
Tâche terminée en 0,016 secondes
Rapport d'erreur -
ORA-20101: division par zéro
ORA-06512: at line 9
Compilateur - Journal
Ligne 11 Colonne 1
Insertion
Modifié
Windows : CF

Exception INVALID_NUMBER et OTHER

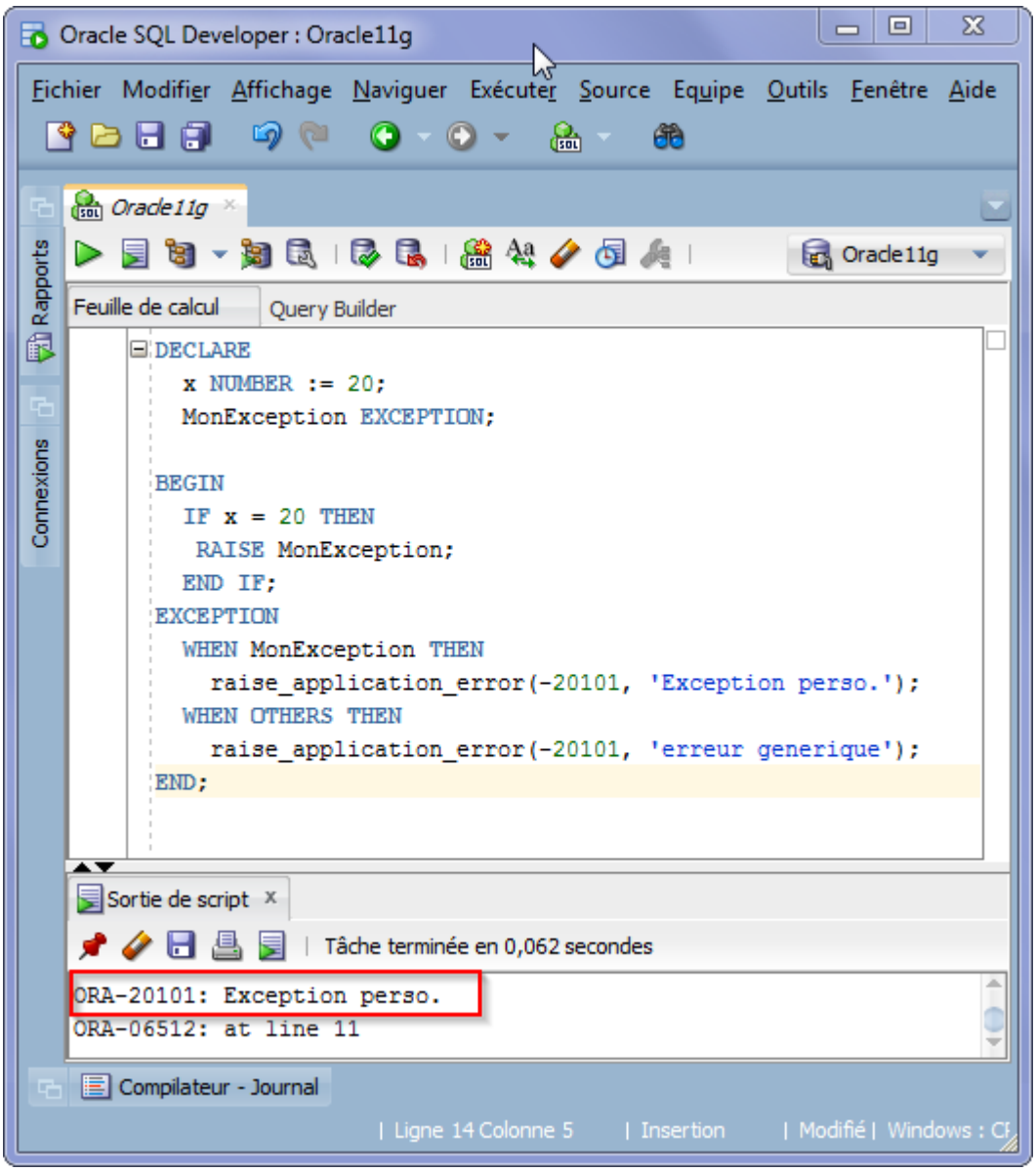


Vous pouvez consulter les autres exceptions (`DUP_VAL_ON_INDEX`, `NO_DATA_FOUND`, `NOT_LOGGED_ON`...) dans le manuel de référence d'Oracle.

10.2 Définir une execption personalisée

En plus des exceptions prédéfinies, Il est possible de définir sa propre exception :

```
DECLARE
  x NUMBER := 20;
  MonException EXCEPTION;

BEGIN
  IF x = 20 THEN
   RAISE MonException;
  END IF;
EXCEPTION
  WHEN MonException THEN
    raise_application_error(-20101, 'Exception
perso.');
  WHEN OTHERS THEN
    raise_application_error(-20101, 'erreur
generique');
END;
```


Oracle SQL Developer : Oracle11g
Fichier Modifier Affichage Naviguer Exécuter Source Equipe Outils Fenêtre Aide
Oracle11g
Rapports
Connexions
Oracle11g
Feuille de calcul
Query Builder
DECLARE
x NUMBER := 20;
MonException EXCEPTION;
BEGIN
IF x = 20 THEN
RAISE MonException;
END IF;
EXCEPTION
WHEN MonException THEN
raise_application_error(-20101, 'Exception perso.');
WHEN OTHERS THEN
raise_application_error(-20101, 'erreur generique');
END;
Sortie de script
Tâche terminée en 0,062 secondes
ORA-20101: Exception perso.
ORA-06512: at line 11
Compilateur - Journal
Ligne 14 Colonne 5
Insertion
Modifié
Windows : Cf

11 Script de création de la base de données (Oracle 11g)

Pour pouvoir faire les exemples de ce manuel, vous devez créer les tables et les peupler en exécutant le script qui suit.

Tous les exemples ont été testés sur Oracle version 11g.

```
ALTER SESSION SET NLS_DATE_FORMAT='DD-MM-YYYY';

CREATE TABLE Employes
(
no_emp               int PRIMARY KEY,
prenom_emp      varchar(20),
nom_emp              varchar(20),
fonction        varchar(20),
salaire         number,
date_embauche   date,
date_naissance  date,
pays            varchar(20)
)
/
-----------------------------------------------
ALTER TABLE Employes
ADD CONSTRAINT CK_date_embauche CHECK (date_embauche >
date_naissance)
/
-----------------------------------------------
INSERT INTO Employes
VALUES(1,'Michel','Tremblay','Vendeur',40000,'23-12-2001','09-02-
1970','Canada')
/
INSERT INTO Employes
VALUES(2,'Olivier','Dupont','Acheteur',38000,'13-10-2003','04-11-
1969','Canada')
/
INSERT INTO Employes
VALUES(3,'Pham','Tran','Analyste',48000,'09-12-1990','06-08-
1960','Coree')  Montreal
/
INSERT INTO Employes
VALUES(4,'Nathalie','Dagenais','Directrice',80000,'22-03-
2000','20-02-1975','Canada')
/
INSERT INTO Employes
VALUES(5,'Eric','Pare','Acheteur',39000,'23-12-2010','01-12-
1985','Canada')
/
```

```
INSERT INTO Employes
VALUES(6,'Marc','Bergeron','Vendeur',42000,'20-11-2012','07-09-
1988','Canada')
/
INSERT INTO Employes
VALUES(7,'Maxime','Laberge','Vendeur',41000,'30-09-2005','19-10-
1983','Canada')  Toronto
/
INSERT INTO Employes
VALUES(8,'Pierre','Lorimier','Acheteur',38800,'15-03-2002','29-
03-1984','Canada')
/
INSERT INTO Employes
VALUES(9,'Valérie','Ward','Analyste',52000,'19-07-2011','27-12-
1990','Canada')
/
INSERT INTO Employes
VALUES(10,'Luc','Picard','Vendeur',34000,'23-01-2008','11-11-
1976','Canada')
/
-----------------------------------------------
ALTER TABLE Employes
ADD ville varchar(20)
/

-----------------------------------------------
CREATE TABLE Fournisseurs
(
no_fournisseur  int PRIMARY KEY,
nom_fournisseur varchar(20),
pays            varchar(20),
ville           varchar(20)
)
/

-----------------------------------------------
INSERT INTO Fournisseurs VALUES(1,'Research In
Motion','Canada','Waterloo')
/
INSERT INTO Fournisseurs VALUES(2,'Samsung','Coree','Séoul')
/
INSERT INTO Fournisseurs VALUES(3,'Apple','États-Unis','Boston')
/

-----------------------------------------------
CREATE TABLE Categories
(
no_categorie    int PRIMARY KEY,
```

```
nom_categorie   varchar(20)
)
/

-----------------------------------------------
INSERT INTO  Categories VALUES(100,'Tablette')
/
INSERT INTO  Categories VALUES(200,'Cellulaire')
/
INSERT INTO  Categories VALUES(300,'Ordinateur portable')
/

-----------------------------------------------
CREATE TABLE Produits
(
no_produit      int PRIMARY KEY,
nom_produit          varchar2(20),
no_categorie    int,
no_fournisseur  int,
prix_unitaire   number,
unites_commandees    int,
unites_en_stock int
)
/

-----------------------------------------------
ALTER TABLE Produits
ADD CONSTRAINT FK_CATEGORIES   FOREIGN KEY(no_categorie)
REFERENCES Categories(no_categorie)
/

ALTER TABLE Produits
ADD CONSTRAINT FK_FOURNISSEURS FOREIGN KEY(no_fournisseur)
REFERENCES Fournisseurs(no_fournisseur)
/

-----------------------------------------------
INSERT INTO  Produits VALUES(1,'Galaxy Tab2',100,2,249,40,9)
/
INSERT INTO  Produits VALUES(2,'Iphone5',200,3,400,39,102)
/
INSERT INTO  Produits VALUES(3,'BlackBerry
Curve',200,1,189,28,23)
/
INSERT INTO  Produits VALUES(4,'PlayBook',100,1,149,23,56)
/
INSERT INTO  Produits VALUES(5,'Galaxy S3',200,2,489,9,81)
/
```

```
INSERT INTO  Produits VALUES(6,'Ipad',100,1,499,43,13)
/
INSERT INTO  Produits VALUES(7,'Samsung 15',300,2,399,8,0)
/

-----------------------------------------------
CREATE TABLE Clients
(
no_client  int PRIMARY KEY,
nom_client varchar(20),
pays       varchar(20),
ville      varchar(20)
)
/

-----------------------------------------------
INSERT INTO Clients VALUES(10,'Future Shop','Canada','Toronto')
/
INSERT INTO Clients VALUES(20,'BestBuy','Canada','Montréal')
/
INSERT INTO Clients VALUES(30,'BureauEnGros','Canada','Montréal')
/
INSERT INTO Clients VALUES(40,'ElectroHola','Mexique','Mexico')
/

-----------------------------------------------
CREATE TABLE Commandes
(
no_commande                int,
no_client             int,
date_envoi            date,
date_commande         date,
ville_livraison       varchar(20),
pays_livraison        varchar(20),
code_postal_livraison varchar(20),
no_employe            int,
a_livrer_avant        date
)
/

-----------------------------------------------
ALTER TABLE Commandes
ADD CONSTRAINT PK_COMMANCES PRIMARY KEY(no_commande)
/
ALTER TABLE Commandes
ADD CONSTRAINT FK_CLIENTS   FOREIGN KEY(no_client)  REFERENCES
Clients(no_client)
/
```

```
ALTER TABLE Commandes
ADD CONSTRAINT FK_EMPLOYES  FOREIGN KEY(no_employe) REFERENCES
Employes(no_emp)
/

-----------------------------------------------
INSERT INTO Commandes VALUES(1,30,'15-04-2002','13-04-
2002','Montréal','Canada','H7R4Y8',3,'20-04-2002')
/
INSERT INTO Commandes VALUES(2,40,'26-12-2010','24-12-
2010','Mexico','Mexique','HOLA',7,'31-12-2010')
/
INSERT INTO Commandes VALUES(3,20,'11-03-2012','09-03-
2012','Montréal','Canada','H7R3B2',1,'15-03-2012')
/
INSERT INTO Commandes VALUES(4,10,'03-12-2012','30-11-
2012','Toronto','Canada','H7R2G1',2,'09-12-2012')
/
INSERT INTO Commandes VALUES(5,40,'23-12-2012','19-10-
2012','Mexico','Mexique','HOLA',5,'29-10-2012')
/
INSERT INTO Commandes VALUES(6,40,'09-08-2003','19-10-
2003','Mexico','Mexique','HOLA',4,'29-10-2003')
/

-----------------------------------------------
CREATE TABLE Details_commandes
(
no_commande          int,
no_produit      int,
prix_unitaire   number,
quantite        int
)
/

-----------------------------------------------
ALTER TABLE Details_commandes
ADD CONSTRAINT PK_DETAILS_COMMANCES PRIMARY
KEY(no_commande,no_produit)
/

ALTER TABLE Details_commandes
ADD CONSTRAINT FK_COMMANDES FOREIGN KEY(no_commande) REFERENCES
Commandes(no_commande)
/

ALTER TABLE Details_commandes
```

```
ADD CONSTRAINT FK_PRODUITS  FOREIGN KEY(no_produit)  REFERENCES
Produits(no_produit)
/

-----------------------------------------------
INSERT INTO Details_commandes VALUES(6,1,249,1)
/
INSERT INTO Details_commandes VALUES(6,3,149,5)
/
INSERT INTO Details_commandes VALUES(6,6,399,2)
/
INSERT INTO Details_commandes VALUES(1,3,149,8)
/
INSERT INTO Details_commandes VALUES(1,4,489,22)
/
INSERT INTO Details_commandes VALUES(2,1,400,13)
/
INSERT INTO Details_commandes VALUES(2,2,189,8)
/
INSERT INTO Details_commandes VALUES(2,5,499,5)
/
INSERT INTO Details_commandes VALUES(3,1,249,3)
/
INSERT INTO Details_commandes VALUES(3,6,399,2)
/
INSERT INTO Details_commandes VALUES(4,4,489,9)
/
INSERT INTO Details_commandes VALUES(5,3,149,24)
/
```