Interface C/SQL

Philippe Rigaux

7 décembre 1999

Table des matières

Programmation avec SQL

Ce chapitre présente l'intégration de SQL et d'un langage de programmation classique (le C en l'occurence). l'utilisation conjointe de deux langages résulte de l'insuffisance de SQL en matière de traitement de données : on ne sait pas faire de boucle, de tests, etc. On utilise donc SQL pour extraire les données de la base, et le C (ou tout autre langage) pour manipuler ces données. La difficulté principale consiste à transcrire des données stockées selon des types SQL en données manipulées par le C.

La présentation reste volontairement très succincte : il s'agit d'illustrer le mécanisme de cohabitation entre les deux langages. Le contenu consiste essentiellement à détailler un programme réel qui peut s'exécuter sous le SGBD Oracle. L'interface SQL/C est normalisée dans SQL2, et l'exemple que l'on va trouver ensuite est très proche de cette norme.

Le SQL dit 'dynamique' qui permet de créer et d'exécuter en direct des ordres SQL n'est pas abordé ici. L'objectif est simplement de montrer comment on se connecte, comment on extrait des données de la base, et de donner quelques indications et conseils sur la gestion des erreurs, la structuration du code et les erreurs à éviter.

1 Un exemple

Pour commencer, voici un exemple complet. Il suppose qu'il existe dans la base une table `Film` dont voici le schéma (voir `film.sql`) :

```
CREATE TABLE film  (ID_film        NUMBER(10) NOT NULL,
                    Titre          VARCHAR (30),
                    Annee          NUMBER(4),
                    ID_Realisateur NUMBER(10));
```

Voici un programme qui se connecte et recherche le film d'id 1. Bien entendu les numéros en fin de ligne sont destinés aux commentaires.

```
#include <stdio.h>

EXEC SQL INCLUDE sqlca;                          (1)

typedef char asc31[31];                          (2)

int main (int argc, char* argv[])
{
  EXEC SQL BEGIN DECLARE SECTION;                (3)

  EXEC SQL TYPE asc31 IS STRING(31) REFERENCE;   (2')

  int        ora_id_film, ora_id_mes, ora_annee;
  char  user_id = '/';
  asc31        ora_titre;                        (2'')

  short vi1, vi2, vi3;                           (4)

  EXEC SQL END DECLARE SECTION;                  (3')
  EXEC SQL WHENEVER SQLERROR goto sqlerror;      (5)

  EXEC SQL CONNECT :user_id;                     (6)

  ora_id_film = 1;                               (7)
  ora_id_mes = 0; ora_annee = 0;
  strcpy (ora_titre,"");

  EXEC SQL SELECT titre, annee, id_realisateur   (8)
  INTO  :ora_titre:vi1, :ora_annee:vi2,
        :ora_id_mes:vi3
  FROM  film
  WHERE id_film = 1;

  printf ("Titre : %s Annee : %d Id mes %d  \n",
   ora_titre, ora_annee, ora_id_mes);

sqlerror: if (sqlca.sqlcode != 0)                (9)
        fprintf (stderr,"Erreur  NO %d : %s\n",
          sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
}
```

Il reste à précompiler ce programme (le SGBD remplace alors tous les `EXEC SQL` par des appels à ses propres fonctions C), à compiler le `.c` résultant de la précompilation, et à faire l'édition de lien avec les librairies pertinentes. Voici les commentaires pour chaque partie du programme ci-dessus.

1. cette ligne est spécifique à Oracle aui communique avec le programme *via* la structure `sqlca`: il faut inclure le fichier `sqlca.h` **avant** la précompilation, ce qui se fait avec la commande `EXEC SQL INCLUDE`.

2. Le principal problème en PRO*C est la conversion des `VARCHAR` de SQL en chaînes de caractères C contenant le fameux `\0`. Le plus simple est de définir explicitement l'équivalence entre un type manipulé par le programme et le type SQL correspondant. Cela se fait en deux étapes :

   (a) On fait un `typedef` pour définir le type du programme : ici le type `asc31` est synonyme d'une chaîne de 31 caractères C.

   (b) On utilise (ligne 2') la commande `EXEC SQL TYPE` pour définir l'équivalence avec le type SQL.

   Maintenant, le SGBD gérera convenablement la conversion des `VARCHAR` vers une variable C (2") en ajoutant le `\0` après le dernier caractère non-blanc.

3. Le transfert entre la base de données et le C se fait par l'intermédiaire de "variables hôtes" qui doivent être déclarées dans un bloc spécifique (3 et 3').

4. Il n'y a pas, en C, l'équivalent de la "valeur nulle" (qui correspond en fait à **l'absence** de valeur). Pour savoir si une valeur ramenée dans un ordre SQL est nulle, on doit donc utiliser une variable spécifique, dite **indicatrice**. Cette variable est toujours de type `short`

5. Dans le cas où une erreur survient au moment de l'exécution d'un ordre SQL, il faut indiquer le comportement à adopter. Ici on se déroute sur une étiquette `sqlerror`.

6. Connexion à la base : indispensable avant tout autre ordre. Ici on utilise la connexion automatique Oracle.

7. Initialisation des variables de communication. **Cette initialisation est indispensable**.

8. Exemple d'un ordre `SELECT`. Pour chaque attribut sélectionné, on indique dans la clause `INTO` la variable réceptrice **suivi de la variable indicatrice**. Attention le SGBD génère une erreur si on lit une valeur nulle sans utiliser de variable indicatrice.

9. Gestion des erreurs : le champ `sqlcode` de la structure `sqlca` et mis à jour après chaque ordre SQL. Quand il vaut 0, c'est qu'on n'a pas rencontré d'erreur. La valeur 1403 (spécifique Oracle) indique qu'aucune ligne n'a été trouvée. Toute valeur négative indique un erreur, auquel cas le message se trouve dans `sqlca.sqlerrm.sqlerrmc`.

A peu près l'essentiel de ce qui est suffisant pour écrire un programme C/SQL se trouve dans le code précédent. La principale fonctionnalité non évoquée ci-dessus est l'emploi de **curseurs** pour parcourir un ensemble de n-uplets. SQL manipule des ensembles, notion qui n'existe pas en C : il faut donc parcourir l'ensemble ramené par l'ordre SQL et traiter les tuples un à un. Voici la partie du code qui change si on souhaite parcourir l'ensemble des films.

```
/* Comme précédemment, jusqu'a EXEC SQL WHENEVER SQLERROR ... */
```

```
EXEC SQL DECLARE CFILM CURSOR FOR
SELECT id_film, titre, annee, id_realisateur
FROM film;

EXEC SQL CONNECT :user_id;

ora_id_film = 0; ora_id_mes = 0;
ora_annee = 0; strcpy (ora_titre,"");

EXEC SQL OPEN CFILM;

EXEC SQL FETCH CFILM INTO :ora_id_film:vi1, :ora_titre:vi2,
                           :ora_annee:vi3, :ora_id_mes:vi4;

while (sqlca.sqlcode != ORA_NOTFOUND)
{
  printf ("Film no %d. Titre : %s Annee : %d Id mes %d  \n",
        ora_id_film, ora_titre, ora_annee, ora_id_mes);

  EXEC SQL FETCH CFILM INTO :ora_id_film:vi1, :ora_titre:vi2,
                           :ora_annee:vi3, :ora_id_mes:vi4;
}
EXEC SQL CLOSE CFILM;

/* Comme avant ... */
```

On déclare un curseur dès qu'un ordre SQL ramène potentiellement plusieurs n-uplets. Ensuite chaque appel à la clause `FETCH` accède à un n-uplet, jusqu'à ce que `sqlca.sqlcode` soit égal à 1403 (ici on a déclaré une constante `ORA_NOTFOUND`).

Comme d'habitude, il est recommandé d'organiser le code avec des fonctions. D'une manière générale, il paraît préférable de bien séparer le code gérant les accès à la base du code implantant l'application proprement dite. Quelques suggestions sont données dans la section suivante.

2 Développement en C/SQL

La recherche d'information dans une table est une bonne occasion d'isoler une partie bien spécifique du code en créant une fonction chargée d'accéder à cette table, de vérifier la validité des données extraites de la base, d'effectuer les conversions nécessaires, etc. De plus une telle fonction a toutes les chances d'être utile à beaucoup de monde.
Les deux cas les plus courants d'accès à une table sont les suivants.

1. **Recherche** d'un n-uplet avec la clé.
2. **Boucle** sur les n-uplets d'une table en fonction d'un intervalle de valeurs pour la clé.

Du point de vue de la structuration du code, voici les stratégies qui me semblent les plus recommandables pour chaque cas.

Recherche avec la clé

1. On définit une structure correspondant au sous-ensemble des attributs de la table que l'on souhaite charger dans le programme.
2. On définit une fonction de lecture (par exemple `LireFilm`) qui prend en entrée un pointeur sur une structure et renvoie un booléen. Au moment de l'appel à la fonction, on doit avoir initialisé le champ correspondant à la clé.
3. Dans la fonction, on exécute l'ordre SQL, on effectue les contrôles nécessaires, on place les données dans les champs de la structure. On renvoie `TRUE` si on a trouvé quelque chose, `FALSE` sinon.

Voici par exemple le squelette de la fonction `LireFilm`.

```
boolean LireFilm (Film *film)
{
  /* Declarations */
  ....
  /* Initialisations */
  ...
  ora_id_film = film->id_film;
  ...
  /* Ordre SELECT  */
  EXEC SQL SELECT ...

  /* Test          */
  if (sqlca.sqlcode == ORA_NOTFOUND) return FALSE;
  else ...

  /* Controles divers et placement dans la structure */
  ...
  film->id_film = ora_id_film;
  ...
  return TRUE;
}
```

Et voici comment on appelle la fonction.

```
  Film film;
  ...
  film.id_film = 34;
  if (LireFilm (&film))
    ... /* On a trouve le n-uplet  */
  else
    ... /* On n'a rien trouve */
```

Donc la fonction appelante ne voit rien de l'interface SQL et peut se consacrer uniquement à la manipulation des données.

Recherche par intervalle

On peut suivre à peu près les mêmes principes, à ceci près qu'il faut :

1. Passer en paramètre les critères de recherche.
2. Gérer l'ouverture et la fermeture du curseur.

Pour le deuxième point on peut procéder comme suit. On place dans la fonction une variable statique initialisée à 0. Au premier appel, cette variable est nulle, et on doit ouvrir le curseur et changer la valeur à 1 avant de faire le premier `FETCH`. Aux appels suivants la valeur est 1 et on peut faire simplement des `FETCH`. Quand on a atteint le dernier n-uplet, on ferme le curseur et on remet la variable à 0. Voici le squelette de la fonction `BoucleFilms` qui effectue une recherche sur un intervalle de clés.

```
boolean BoucleFilms (Film *film, int cle_min, int cle_max)
{
 /* Declarations des variables et du curseur, initialisations ... */
 ...
 static debut = 0;
 ...

 /* Test d'ouverture du curseur      */
 if (debut == 0)
 {
   EXEC SQL OPEN ...
   debut = 1;
 }

 /* Dans tous les cas on fait le FETCH     */
 EXEC SQL FETCH ...

 if (sqlca.sqlcode == ORA_NOTFOUND)
 {
   EXEC SQL CLOSE ...
   debut = 0;
   return FALSE;
 }
 else
 {
   /* Faire les contrôles et placer les données dans film */
   ...
   return TRUE
 }
}
```

Voici comment on utilise cette fonction.

```
Film film;
int cle_min, cle_max;
...
while (BoucleFilms (&film, cle_min, cle_max))
{
  ....
}
```

Notez qu'avec l'utilisation combinée des fonctions et des structures, non seulement on clarifie beaucoup le code, mais on rend très facile l'ajout d'une nouvelle donnée. Il suffit de modifier la structure et l'implantation de la fonction de lecture. Tout le reste est inchangé.

3 Compléments

Voici, à titre de complément, les principales fonctionnalités dáccès à une base de données et leur expression en C/SQL.

3.1 Validation et annulation

1. Validation : `EXEC SQL COMMIT WORK;`
2. Anulation : `EXEC SQL ROLLBACK WORK;`

Si on ne fait pas de `COMMIT` explicite, Oracle effectue un `ROLLBACK` à la fin du programme.

3.2 `UPDATE, DELETE, INSERT`

On utilise ces commandes selon une syntaxe tout à fait semblable à celle du `SELECT`. Voici des exemples.

```
/* Les variables ora_... sont déclarées comme précédemment */

EXEC SQL INSERT INTO film (id_film, titre, annee, id_mes)
VALUES   (:ora_id_film, :ora_titre, :ora_annee, :ora_mes);
...
EXEC SQL DELETE FROM film
WHERE id_film = :ora_id_film;
...
EXEC SQL UPDATE film SET annee = :ora_annee, id_mes=:ora_id_mes
WHERE  id_film = :ora_id_film;
```

3.3 Valeurs nulles

On teste les valeurs nulles avec les variables indicatrices (voir ci-dessus). Une valeur de -1 après l'exécution d'un `SELECT` indique que la valeur extraite de la base est nulle (spécifique Oracle).

```
#define ORA_NULL -1
```

```
...
EXEC SQL SELECT ... INTO :ora_id_mes:vi ...
...
if (vi == ORA_NULL)
  /* L'identifiant du metteur en scene est inconnu */
...
```