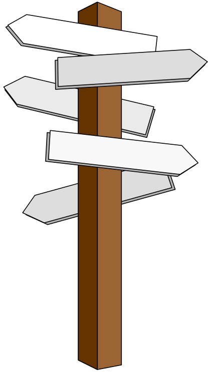
IUP MIAGE
UNIVERSITE PARIS X NANTERRE

Cours n°7 Programmation - Introduction au mode application

Chantal Reynaud

Université Paris X - Nanterre UFR SEGMI - IUP MIAGE



Plan

I. Introduction
II. Le mode module
III. Le mode intégré
IV. SQL dynamique

Partie I. Introduction

- Un langage autonome comme SQL est insuffisant à lui seul pour écrire des applications accédant à une base de données. On écrit alors des programmes associant des commandes SQL et des instructions d 'un langage de programmation classique, appelé *langage hôte*.
- Il existe différents modes d 'utilisation de SQL :

  - le mode **interactif** (utilisation via SQL*Plus : peu utilisable par un utilisateur final car la syntaxe est difficile.

  - le mode **module** (modular) ou **intégré** (embedded) : mode utilisé pour l 'extraction et la manipulation de données dans des programmes de gestion exécutant *régulièrement* certaines opérations.

  - le mode **dynamique** : mode qui permet de disposer de facilités pour traiter des requêtes qui *ne sont pas connues à l 'avance*.

Différentes méthodes d 'écriture d 'une application basée sur SQL normalisées par l 'ANSI ISO au travers de la norme SQL-92.

- via l 'utilisation d '**API** dans les environnements clients-serveurs.

Partie 2. Le mode module

- Technique aujourd 'hui peu utilisée
- Un module est une unité compilable (fichier séparé) composée d'un ensemble de procédures SQL. Dans SQL2, une procédure est une (une seule) commande SQL contenant des paramètres. Les procédures sont appelées à partir d 'un langage de programmation classique. En principe, il y a un module SQL par unité de compilation du langage hôte.
- Difficulté principale : correspondance entre les types de données SQL et ceux du langage hôte.

Etapes successives du développement d'une application :

- Ecriture du programme source dans le langage hôte avec appel aux procédures SQL externes,
- Ecriture du programme source du module SQL,
- Compilation du programme source,
- Application du processeur de module au source SQL,
- Edition des liens entre les 2 modules objet et la librairie SQL,
- Exécution du programme.

UPX IUP MIAGE UNIVERSITE PARIS X NANTERRE

Syntaxe du langage de *module SQL*

Définition d ’un module

MODULE nom_du_module **NAMES ARE** character_set
LANGUAGE C
SCHEMA nom_du_schema **AUTHORIZATION** identification_du_createur
Déclarations de tables temporaires utilisées dans les procédures
Déclarations des curseurs utilisées dans les procédures
Procédures

Définition d ’une procédure

PROCEDURE nom_procedure (liste_def_parametre)
instruction_SQL

def_parametre ::= nom_parametre type_SQL | SQLSTATE | SQLCODE
SQLSTATE et SQLCODE servent à la détection des erreurs
Les autres paramètres sont toujours préfixés par « : »

Exemples de procédures

PROCEDURE update_prix_article
(**SQLSTATE**, :nouveau_prix INTEGER, :numero_article CHAR(8))
UPDATE articles
SET art_pv = :nouveau_prix
WHERE art_num = :numero_article;

PROCEDURE lecture_nom_et_prix_article
(**SQLSTATE**, :numero_article CHAR(9), :nom_article VARCHAR(25), :pv_article INTEGER)
SELECT art_nom, art_pv
INTO :nom_article, :pv_article
FROM articles
WHERE art_num = :numero_article;

IUP MIAGE
UNIVERSITE PARIS X NANTERRE

Exemple de code C utilisant un module SQL

```
...
int prix;
char numero[9];
char retcode[6];
…
gets(numero);
scanf(" %d " ), &prix);
…
update_prix_article(retcode, prix, numero);
if(strcmp(retcode, "00000")) gestion_erreur(retcode);
lecture_nom_et_prix_article(retcode, numero, &prix);
if(strcmp(retcode, "00000")) gestion_erreur(retcode);
printf("L'article n° %s est maintenant au prix %d\n", numero, prix);
...
```

Av. : séparation nette des langages

Inc. : répétitions des déclarations

Partie 3. Le mode intégré

- Cette approche, fréquemment utilisée dans les environnements non clients-serveurs, est proposée par tous les SGBD.
- Les instructions SQL sont placées directement dans le texte du langage hôte. Elles commencent par **EXEC SQL** et se terminent par ";"
- Toute instruction SQL interactive valide est utilisable dans un programme.
- Présence d'instructions **déclaratives/exécutables**.

Etapes successives de l'écriture et de l'exécution d'un programme :

- écrire le programme source en incluant les instructions SQL,
- précompiler le programme source pour obtenir un programme source modifié,
- compiler le programme source modifié,
- éditer les liens en incluant la librairie SQL,
- exécuter le programme.

Les variables hôtes

- Les instructions SQL peuvent faire référence aux variables du langage hôte et inversement une variable du langage hôte peut se mettre à la place d'une constante dans les instructions SQL.
- Les variables hôtes sont déclarées à l'aide d'instructions du langage hôte.
- Dans les instructions SQL, elles sont préfixées de ":". En dehors des instructions SQL, ce sont de simples variables.
- Selon la façon dont elles sont utilisées, on parle de variables d'hôte d'entrée (entrées de valeurs dans la base) ou de sortie.

Attention : pas de variables hôtes dans les expressions de définition des données : ALTER, DROP, CREATE.

Exemple :

EXEC SQL SELECT gerant_mag
INTO *:gerant_mag*
FROM magasin
WHERE num_mag = *:num_mag*;

Place le résultat de la requête dans une variable du langage hôte

Remarque : Les noms de variables et de colonnes peuvent être identiques.

Les variables hôtes

Exemple d'utilisation dans un programme

```
…
main ()
{
EXEC SQL BEGIN DECLARE SECTION;
int  emp_number;
char temp[20];
EXEC SQL END DECLARE SECTION;
...
/* saisie des valeurs des variables hôtes d'entrée */
printf("Numéro d'employé ?")
gets(temp);
emp_number = atoi(temp);
printf("Nom d'employé ?")
gets(emp_name.arr);
emp_name.len = strlen(emp_name.arr);

EXEC SQL INSERT INTO EMP (empno, ename)
VALUES (:emp_number, :emp_name);
...
}
```

Les variables indicatrices

• Les variables indicatrices sont associées à des variables hôtes. A chaque fois qu'une variable hôte est utilisée dans une expression SQL, un code résultat est stocké dans sa variable indicatrice associée.

• Il n'existe pas, en C, l'équivalent de NULL (absence de valeur). Pour savoir si une valeur ramenée dans un ordre SQL est nulle, on doit utiliser une variable spécifique, dite *indicatrice*. Cette variable est toujours de type *short*.

• L'utilisation de variables indicatrices est facultatif mais le SGBD génère une erreur si on lit une valeur NULL sans variable indicatrice.

1

```
...
EXEC SQL BEGIN DECLARE SECTION;
...
short vi1;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT gerant_mag
INTO :gerant_mag: vi1
FROM magasin
WHERE num_mag = :num_mag;
```

Variables en entrée	
-1	Valeur NULL
>= 0	Valeur de la variable hôte assignée à la colonne
Variables en sortie	
-1	NULL dans la colonne, valeur de la variable hôte indéterminée
0	Valeur de la colonne assignée
> 0	A assigné une valeur tronquée
-2	La valeur de la colonne originale n'a pu être déterminée

2

```
set ind_comm = -1;
EXEC SQL INSERT INTO emp(empno, comm) VALUES (:emp_number, :commission:ind_comm);
```

➡ NULL peut de cette façon être stocké dans la colonne COMM.

Equivalence entre les types C et les types Oracle

Types C	Types oracle externes
char ou char[n]	VARCHAR2
char *	CHARZ
int ou int*	INTEGER
short ou short *	INTEGER
long ou long *	INTEGER
float ou float *	FLOAT
double ou double *	FLOAT

- Les variables hôtes d'un programme SQL intégré doivent posséder un type compatible avec un type SQL.
- Le principal problème en PRO*C est la conversion des VARCHAR de SQL en chaînes de caractères C contenant \0. Une solution est de définir explicitement l'équivalence :

```
typedef char asc31[31];
...
EXEC SQL TYPE asc31 IS CHARZ(31) REFERENCE;
…
asc31          titre;
```

Notion de curseur

• L'instruction SELECT pose un problème particulier lorsque le résultat est une table contenant plus d'une ligne. Le résultat de la requête doit être exploité par le langage hôte mais celui-ci n'est pas adapté pour traiter plus d'un enregistrement à la fois. On utilise alors la notion de curseur.

•Un curseur est une zone temporaire dans laquelle sont stockées les tuples ramenés par un SELECT. On ne peut lire cette zone que séquentiellement. En pratique, la requête est d'abord exécutée. Un curseur est associé au résultat de la requête qui est conservé en mémoire. Grâce au curseur, on peut se déplacer dans le résultat à partir du langage hôte.

•L'utilisation d'un curseur passe par 4 phases :

DECLARE : déclaration du curseur
OPEN : ouverture du curseur càd exécution de la requête et création de la table résultat
FETCH : extraction des données du curseur ligne par ligne
CLOSE : fermeture du curseur, càd libération des ressources allouées.

Séquence typique d'expressions de gestion d'un curseur

IUP MIAGE
UNIVERSITE PARIS X NANTERRE

/* definition d'un curseur */

EXEC SQL DECLARE emp_cursor **CURSOR FOR**
SELECT ename, job
FROM emp
WHERE empno = :emp_number
FOR UPDATE OF job;

/* ouverture d'un curseur et identification de l'ensemble des tuples actifs */

EXEC SQL OPEN emp_cursor;

/* break si le dernier tuple est déjà lu */
EXEC SQL WHENEVER NOT FOUND break;

/* boucle de lecture et traitement */
for (;;)
{
EXEC SQL FETCH emp_cursor **INTO** :emp_name, :job_title;

/* Traitement optionnel des données lues */
EXEC SQL UPDATE emp
SET job = :job_title
WHERE CURRENT OF emp_cursor;

EXEC SQL FETCH emp_cursor **INTO** :emp_name, :job_title;

}
…
/* fermeture du curseur */
EXEC SQL CLOSE emp_cursor;
...

La gestion des erreurs (*avant SQL2*)

• Utilisation de la variable *sqlcode* (= 0 si l'instruction s'est bien déroulée, = 1403 sous Oracle quand aucune ligne n'a été trouvée, < 0 si erreur).

sqlcode est un champ de la structure SQLCA (SQL Communication Area). On inclut cette structure dans le source avec l'instruction :

```
EXEC SQL INCLUDE SQLCA;
```

Pour accéder au code erreur, on utilise alors : *sqlca.sqlcode*.
Le message de l'erreur se trouve dans *sqlca.sqlerrm.sqlerrmc*

• Conjointement, on utilise l'instruction :

```
EXEC SQL WHENEVER situation action;
```

situation = SQLERROR ou NOT FOUND ou SQLWARNING
action = action entreprendre = STOP, CONTINUE, GOTO ou DO (dans Oracle)

La gestion des erreurs (*avant SQL2 : exemple*)

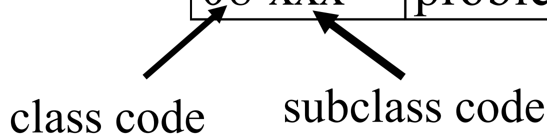
```
…
EXEC SQL INCLUDE sqlca;
…
main ()
{
EXEC SQL BEGIN DECLARE SECTION;
…
EXEC SQL END DECLARE SECTION;
EXEC SQL WHENEVER SQLERROR goto sqlerror;
…
….
sqlerror:  if (sqlca.sqlcode != 0)
     printf ("erreur no %d : %s\n",
             sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
}
```

La gestion des erreurs (*depuis SQL2*)

• **sqlstate**

Exécution erronée si le code de la classe sqlstate est différent de 00, 01 ou 02

sqlstate	**signification**
00 000	exécution avec succès
01 xxx	avertissement
02 xxx	pas de ligne affectée
22 xxx	pb dans le format ou la conversion des données
42 xxx	erreur de syntaxe
08 xxx	problème de connexion à une base



• **diagnostics area :** structure que le programme peut consulter pour obtenir des informations détaillées sur le déroulement des instructions. Elle comporte une partie fixe (header area) et un nombre quelconque de zones de détails (details area).

La gestion des erreurs (*Oracle*)

• WHENEVER **DO**

•sqlerrm.sqlerrmc de la structure sqlca ne contient que les 70 premiers caractères du message d'erreur. Pour pallier ce problème, Oracle propose d'utiliser la fonction **sqlglm** pour extraire in extenso le message :

> void **sqlglm** (char *message, size_t * taille_max, size_t * lg_reelle)
> message est un tampon dans lequel Oracle copie le message d'erreur, taille_max indique la taille maximale du tampon, lg_reelle est initialisé par Oracle avec la longueur réelle du message.

Ex : EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error");

```
void sql_error(msg)
char *msg;
{
char err_msg[128];
int buf_len, msg_len;

EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK RELEASE;
printf("\n%s\n", msg);
buf_len = sizeof(err_msg);
sqlglm(err_msg, &buf_len, &msg_len);
printf("%.s\n", msg_len, err_msg);
exit(1);
}
```

UPX IUP MIAGE UNIVERSITE PARIS X NANTERRE

Autres commandes SQL

• Connexion à la base de données

EXEC SQL CONNECT :username **IDENTIFIED BY** :password;

• Validation

EXEC SQL COMMIT WORK;

• Annulation

EXEC SQL ROLLBACK WORK;

Sans COMMIT, Oracle effectue un ROLLBACK à la fin du programme.

Partie 4 : SQL dynamique

Instructions autres que SELECT

Aucun paramètre

- Instructions possibles : UPDATE, INSERT, DELETE + les instructions du LDD et du LCD.

Exemples d'instructions :
1) DELETE FROM emp WHERE deptno = 20
2) GRANT SELECT on emp to 'Scott'

```
…
gets(instruction);
EXEC SQL EXECUTE IMMEDIATE :instruction;
```

- L'instruction est lue interactivement au clavier sous forme d'une chaîne de caractères. EXECUTE IMMEDIATE permet de l'analyser et de l'exécuter. Une variable hôte de type chaîne de caractères est nécessaire pour contenir l'instruction.

Instructions autres que SELECT

Nombre et type de paramètres connus

```
…
gets(instruction);
EXEC SQL PREPARE i1 FROM :instruction;
for (;;)
{
        lecture des paramètres
        EXEC SQL EXECUTE i1 USING :parametres;
}
```

<u>Ex :</u>

```
gets(instruction);
EXEC SQL PREPARE i1 FROM :instruction;
for (;;)
{
printf("numero de l'article ?");
scanf("%d", &num_article);
printf("quantite restant en stock ?");
scanf("%d", &quantite);

EXEC SQL EXECUTE instruction USING :quantite, :num_article;
…
}
```

Instructions SELECT

Structure du résultat connu - Nombre et type des paramètres connus

Ex :

```
EXEC SQL BEGIN DECLARE SECTION;
char nom[21];
int salaire;
int critere = 10;
EXEC SQL END DECLARE SECTION;

strcpy(instruction, "SELECT nom, salaire FROM employes WHERE critere = :v1");

printf("v1 = %d\n", critere);
printf("\nEmploye\n);
printf("------------\n");

EXEC SQL PREPARE i1 FROM :instruction;

EXEC SQL DECLARE c1 CURSOR FOR i1;
EXEC SQL OPEN c1 USING :critere;
EXEC SQL FETCH c1 INTO :nom, :salaire;
for (;;)
{
        printf("%20s   %d\n, nom, salaire);
        EXEC SQL FETCH c1 INTO :nom, :salaire;
}

EXEC SQL CLOSE c1;
...
```

Instructions SELECT

Structure du résultat inconnu - Nombre de paramètres inconnu

• Difficulté : le nombre de colonnes et leur type ne sont connus qu'à l'exécution. On ne peut donc faire correspondre à chaque colonne lue une variable hôte. Un nombre variable de paramètres entraîne les mêmes difficultés.

• Solution : utilisation de la structure **SQLDA en sortie pour la clause SELECT** et une autre structure **SQLDA en entrée pour les paramètres.** Tout le problème est de travailler avec des structures de taille suffisante. Si on ne tient pas compte des apports de SQL2 (majorité des SGBD actuels), il incombe au programmeur de prendre en charge l'allocation dynamique de la mémoire pour contenir ces structures.

Instructions SELECT

Structure du résultat inconnu - Nombre de paramètres inconnu

Structure générale du programme

```
EXEC SQL DECLARE S STATEMENT;
…
main ()
{
EXEC SQL BEGIN DECLARE SECTION;
    char instruction[1024];
    EXEC SQL VAR instruction IS STRING(1024);
EXEC SQL END DECLARE SECTION;

int nbcolon = 0, nbparam = 0;

SQLDA *bind_desc;
SQLDA *select_desc;
…
saisir_instruction(instruction);

rc = analyse_inst(instruction, &bind_desc, &select_desc, &nbparam, &nbcolon);
…
```

Instructions SELECT

Structure du résultat inconnu - Nombre de paramètres inconnu

(suite)

```
if(nbparam > 0)
        rc=get_param(bind_desc);
EXEC SQL DECLARE C CURSOR FOR S;
EXEC SQL OPEN C USING DESCRIPTOR bind_desc;

if (nbcolon > 0)
        rc = execute_select(select_desc);

free_mem(bind_desc → I[i]);
free_mem(select_desc → V[i]);
sqlsqldafree(sql_single_rctx, bind_desc);
sqlsqldafree(sql_single_rctx, select_desc);

EXEC SQL CLOSE C;
EXEC SQL COMMIT WORK RELEASE;
return;
}
```