8. Вграден SQL

Езикът SQL е език за достъп до БД, който се реализира в два режима: интерактивен и от приложна програма (ПП). В по-голямата си част той е еднакъв в двата варианта. Ще разгледаме накратко основните особености на програмния SQL.

Защо има програмен SQL? SQL е подезик за работа с БД, но не е пълноценен език за програмиране. За потребителите е необходимо да се създадават приложни програми, които работят с БД и имат удобен потребителски интерфейс: екранни форми, менюта, прозорци, изходи във формата на нужните документи.

В промишлените СУБД се предлагат два начина за използване на SQL оператори в програма на език за програмиране.

Вграден SQL (Embedded SQL)

Под вграден SQL (ESQL) се разбира използването на операторите на SQL непосредствено в програма, написана на език за програмиране, наричан базов език. В ролята на базов език може да е език за програмиране с общо предназначение, най-често това е някой от езиците: C, C++, Cobol, Ada, Fortran или език, разработен специално за приложения с БД. Например, в Informix са реализирани ESQL/C, Ada, Cobol, Fortran, същите езици се поддържат и в Oracle. В Informix е реализиран и език наречен 4GL (4 Generation Language).

SQL API (Application Program Interface)

Интерфейсът за програмиране на приложения представлява набор от функции (API-функции), чрез които ПП се обръща към СУБД като й предава SQL оператори за изпълнение или получава резултата от изпълнения оператор. Например такъв интерфейс е реализиран в Oracle за езика C, така наречения OCI (Oracle Call Interface).

8.1. Основни принципи на ESQL

Основната идея на този подход е непосредственото обединение на ESQL операторите с тези на базовия език в една програма. Основните принципи при това обединение са:

- Изходният текст на ПП е написан на смес от два езика - базовия език и ESQL. Следователно е необходим **ESQL препроцесор** (за всеки базов език различен). Той преобразува изходния текст на ПП като компилира ESQL операторите в обръщения към функции, осигуряващи връзка със СУБД. Резултатът от работата на препроцесора е ПП, написана на чист базов език. След това работи стандартният компилатор на базовия език

Следователно е необходимо в изходния текст да се различават конструкциите, които ще бъдат обработени от препроцесора. Всеки ESQL оператор в ПП започва със спецификатор и завършва с ограничител. Нека базовият език е C.

```
EXEC SQL текст-на-esql-оператор ;
```

- Всеки SQL оператор, използван в интерактивен режим, може да се използва и в ESQL. Има обаче някои особености и допълнения при изпълнението на SQL оператор от ПП.
- В ESQL операторите могат да се използват програмни променливи, ще ги наричаме **базови променливи**. Те осигуряват двупосочна връзка между ПП и СУБД.
- Има допълнителни средства за обработка на грешки при ESQL оператори.
- В ESQL има допълнителни оператори, които се използват при извличане на данни от БД. Те осигуряват мост между табличната логика на `SELECT` оператора и позаписната в езиците за програмиране.

8.2. Базови променливи

Базовата променлива е програмна променлива, която се декларира със средствата на базовия език и се използва след това в ESQL оператори. В зависимост от мястото в оператора, където се използват, те биват:

- входни - предават информация от ПП към СУБД
- изходни - предават информация от СУБД към ПП.

- **Къде може да се използва базова променлива?**

Входни променливи може да има на всяко място в ESQL оператор, където е разрешена константа:

- В изразите на условието във фразите `WHERE`, `HAVING`;
- В изразите на фразата `SET` на `UPDATE`;
- Във фразата `VALUES` на `INSERT`;
- Не може да се използва променлива за име на таблица или колона.

Изходни променливи се използват за получаване на данни, извлечени от БД, в операторите `SELECT` и `FETCH`. Има нова фраза `INTO` *списък-базови-променливи*.

- **Обявяване на базова променлива**

Базовите променливи се декларират по правилата на базовия език, но декларацията им е отделена в специална секция.

```
EXEC SQL BEGIN DECLARE SECTION;
      декларации-на-променливи
EXEC SQL END DECLARE SECTION;
```

Това се прави, за да може препроцесорът правилно да компилира ESQL операторите, които използват базови променливи. В ESQL операторите пред името на базовата променлива се слага двоеточие. Така препроцесорът различава имена на променливи от имена на колони и таблици.

- **Базови променливи и `NULL` значения.**

В повечето езици за програмиране, за разлика от SQL, няма значение `NULL`. Ако `SELECT` върне значение `NULL`, как то да се предаде на ПП в изходни променливи и да се обработи от нея. Обикновено се използва допълнителна променлива-индикатор. Ако в определена колона `SELECT` може да върне `NULL`, то се предвиждат две базови променливи, използвани съвместно за тази колона за:

- значението на колоната, ако е определено;
- признак за определеност на значението - 0(определено) и -1(неопределено).

Такава двойка базови променливи могат да се използват и в оператори `UPDATE` и `INSERT`, ако значението, което се въвежда в БД, се съдържа в базова променлива и трябва да е `NULL`.

8.3. Обработка на грешки

При интерактивен режим, информация за изпълнението на SQL опреатор се съобщава на потребителя чрез съобщение на екрана.

- Как е завършил операторът - с грешка или успешно.
- Ако е успешно и операторът е бил `INSERT`, `DELETE` и `UPDATE` - колко реда са добавени, изтрити, изменени.

При вграден SQL, ПП отговаря за обработка на грешките. За тази цел е необходимо да получава нужната информация от СУБД. Това става чрез глобални променливи, които се зареждат от СУБД при изпълнението на оператор и след завършването му се ползват от ПП.

В стандартите е включена така наречената област за връзки - `SQLCA` (SQL Communications Area). Тя представлява структура (в смисъла на езика C), която съдържа променливи с информация за резултата от оператора. Най-важният елемент

на SQLCA е променливата SQLCODE, която съдържа код за завършване на оператора и се поддържа от всички СУБД. Прието е значенията на SQLCODE, които са цели числа да се тълкуват така:

- 0 - успешно изпълнение на оператора;
- отрицателно число - сериозна грешка и операторът не е изпълнен;
- положително число - изключителна ситуация, за която трябва да се съобщи на ПП. Напр., при SELECT не е извлечен нито един ред.

Самите номера на грешки и изключителни ситуации не са стандартизирани. Всяка СУБД си има собствен списък с кодове, които променливата SQLCA.SQLCODE може да върне.

С цел стандартизиране на кодовете за грешки, в SQL2 стандарта е включена нова глобална променлива SQLSTATE, чиито значения са определени в стандарта. СУБД трябва да съобщава за грешки и чрез двете променливи.

За да се опрости обработката на грешки в ПП в много СУБД е добавен нов ESQL оператор, който е директива за препроцесора.

```
WHENEVER {SQL ERROR|NOT FOUND} {CONTINUE|GOTO етикет}
```

8.4. Курсор и ESQL оператори за работа с курсори

Оператор SELECT връща множество редове, а езиците за програмиране нямат средства за обработка на множество записи/структури. Проблемът при SELECT е: Как на ПП да се предаде множество от редове в базови променливи и тя да ги обработи? За тази цел в ESQL се въвежда нов обект - **курсор** и оператори за работа с него.

Едноредов SELECT

Да започнем с по-простия случай, когато SELECT винаги връща най-много един ред. В оператора SELECT е добавена нова фраза - INTO.

```
SELECT целеви-списък
INTO списък-базови-променливи
FROM таблици
[WHERE условие] [GROUP BY фраза] [HAVING фраза] [ORDER BY фраза]
```

По брой и тип променливите във фразата INTO трябва да съответстват на целевия списък (плюс променливи-индикатори, ако се очаква NULL). На всяка променлива се присвоява значението на съответната колона от извлечения ред и SQLCODE е 0 а SQLSTATE е "00000".

Ако SELECT извлече 0 реда, то SQLCODE е 100, а SQLSTATE е "02000". Ако извлече повече от 1 ред, това при този оператор се счита за грешка и SQLCODE е отрицателно число.

Многоредов SELECT - курсор

Ако се очаква SELECT да върне повече от 1 ред, тогава извличането трябва да става с курсор. **Курсорът** представлява нещо като указател, който се движи по редовете от определено множество от извлечени редове. Всеки курсор има идентификатор и с него е свързан оператор SELECT. Това става при деклариране на курсор с оператор DECLARE.

След като курсорът е деклариран, той може да бъде активизиран чрез оператор OPEN. Тогава с него се свързва **активно множество** - множество от редове, извлечени от БД при изпълнението на съответния оператор SELECT. От този момент курсорът представлява указател, който сочи към ред от активното множество, който е достъпен за ПП. Движейки курсора по активното множество ПП може да обработи всички редове от него един по един, колкото и много да са на брой.

Стъпките при работа с курсор са следните:

1. **Деклариране на курсор** - определя се името на курсора и SELECT оператора, свързан с него.(DECLARE)
2. **Активизиране на курсор** - създава се активното множество, като се изпълнява SELECT оператора (започва изпълнението му) и се извършва начално позициониране (обикновено преди първия ред в активното множество). (OPEN)
3. **Позициониране на курсор** - изменя се позицията на курсора, като се насочва към (пореден) ред от активното множество и текущият ред се извлича в базови променливи. (FETCH)
4. **Освобождаване на курсор** - освобождава се активното множество и други ресурси. (CLOSE и FREE)

Новите оператори, добавени в ESQL на Informix, с които се работи с курсор, са: DECLARE, OPEN, FETCH, CLOSE и FREE.

```
DECLARE име-на-курсор [SCROLL] CURSOR
      FOR оператор-select [FOR UPDATE [OF списък-от-колони ]]
```

Този оператор е директива на препроцесора. Освен името на курсора и свързания с него *оператор-select*, тук се определя и типа му:

- **Обикновен (последователен за четене) курсор**. Позволява последователно извличане на редовете от активното множество и само едно преминаване през активното множество.
- **Курсор за обновяване**. Позволява последователно извличане на редовете от активното множество и евентуално изменение или изтриване на текущия ред. По премълчаване се предполага, че курсорът е само за четене.
- **SCROLL курсор**. Позволява извличане на редовете в активното множество в произволен ред и многократно преминаване през него.

Съществуват някои ограничения за *оператор-select* в зависимост от типа на курсора. Ако курсорът е за обновяване, то SELECT операторът трябва:

- да е върху една таблица;
- да не съдържа ключовата дума DISTINCT;
- да не съдържа агрегатни функции;
- да не съдържа фрази GROUP BY, HAVING и ORDER BY;
- да не съдържа UNION.

```
OPEN име-на-курсор
```

Определя активното множество на курсора, като замества базовите променливи в SELECT оператора и започва изпълнението му. Позиционира курсора преди първия ред от активното множество.

При успех SQLCODE е 0, ако се открие грешка в оператора SQLCODE е отрицателно число. Ако активното множество е празно, то SQLCODE е 100, а SQLSTATE е "02000".

Действителната работа, която се извършва при OPEN зависи от типа на курсора и сложността на SELECT оператора. При проста заявка може само да се започне изпълнението на оператора, за да се установи има ли редове в активното множество и да се локализира първия му ред. При сложна заявка (групиране, сортиране и др.) активното множество се създава и записва във временна таблица. Кой от двата метода ще бъде използван зависи от оптимизатора на заявките, следователно ПП не може да знае. При SCROLL курсор се създава временна таблица, в която след това ще се записват извличаните редове.

```
FETCH [NEXT|PREVIOUS|FIRST|LAST|RELATIVE n|ABSOLUTE n|CURRENT]
      име-на-курсор [INTO списък-от-променливи ]
```

Позиционира курсора на искания ред от активното множество и извлича значенията от текущия ред в базовите променливи от фразата INTO. Фразата INTO може да е указана тук или в SELECT оператора.

Ако успешно е извлечен ред, то SQLCODE е 0. Ако е достигнат край на активното множество или то е празно, то SQLCODE е 100.

При курсор за обновяване могат да се използват два специални позиционни варианти на UPDATE и DELETE. Те винаги действат върху един ред - текущия според курсора в момента. Тези оператори дават възможност ПП да преглежда редовете от активното множество по един и ако потребителят реши да обнови само текущия ред.

```
UPDATE име-на-таблица SET фраза
WHERE CURRENT OF име-на-курсор
```

Този оператор не изменя позицията на курсора.

```
DELETE FROM име-на-таблица
WHERE CURRENT OF име-на-курсор
```

След този оператор курсорът сочи между редовете, след изтрития ред.

```
CLOSE име-на-курсор
```

Освобождава активното множество на курсора и ресурсите, отделени за него на сървера. Освобождават се всички ключалки, получени при работа с него. След този оператор може да се изпълни само OPEN или FREE за курсора *име-на-курсор*.

```
FREE име-на-курсор
```

Освобождава всички ресурси, отделени за курсора на сървера и в ПП.

Курсори и транзакции

При използване на курсор за извличане и обновяване, ПП вижда само един ред (текущия ред, извлечен с последния FETCH) във всеки един момент след активизирането му. В същото време конкурентни ПП могат да искат да изменят други редове от активното множество. Това може да създаде проблеми с цялостността на данните.

Ако се използват транзакции в ПП е в сила следното правило: в края на транзакцията автоматично се затварят всички курсори (има едно изключение за HOLD курсори, които не сме разглеждали) и се освобождават всички ключалки. Операторите OPEN, FETCH, DELETE, UPDATE, CLOSE трябва да са в транзакцията.

Правилата за заключване зависят от типа на курсора, дали се използват транзакции и нивото на изолация.

Ако курсорът е обикновен или SCROLL курсор (само за четене) там определящо е нивото на изолация. При SCROLL курсор и БД с тразакции, ПП може да си осигури по-добра изолация, чрез ниво REPEATABLE READ или заключване на цялата таблица. В Informix има едно допълнително ниво CURSOR STABILITY, което може да се използава в БД с транзакции. При извличане на ред (FETCH), той се заключва с SHARE ключалка, която се освобождава при смяна на текущия ред. При БД без транзакции не съществува друг вариант освен ниво READ UNCOMMITTED.

Ако курсорът е за обновяване, активното множество може да се изменя, затова тогава се използват повече ключалки. При извличане на ред (FETCH), той се заключва с promotable ключалка. Това е друг вид ключалка, която гарантира, че други конкурентни ПП могат да поставят само SHARE ключалка. При изпълнение на позиционен DELETE или UPDATE, promotable ключалката се повишава до EXCLUSIVE ключалка. Изменен ред остава заключен до CLOSE или до края на транзакцията, ако се работи с тразакции. Ако редът не е изменен, ключалката може да остане или да се

освободи при смяна на текущия ред, в зависимост от нивото на изолация (при `REPEATABLE READ` остава).

Следват няколко примера написани на Informix - ESQL/C за БД-студенти.

Пример. Програмата чете факултетен номер, извлича името, адреса и курса на студента от БД и ги извежда на екрана.

```
#include <stdio.h>
EXEC SQL define NAME_LEN 30;

main()
{
EXEC SQL include sqlca;
EXEC SQL BEGIN DECLARE SECTION;
    char  fn[7];
    char  ime[NAME_LEN + 1];
    char  adres[NAME_LEN + 1];
    short adres_ind;
    short kurs;
EXEC SQL END DECLARE SECTION;

    printf( "DEMO1 Sample ESQL Program running.\n\n");
    EXEC SQL connect to 'dbstudent';

    printf("Enter FN: ");
    scanf("%s", fn);

    EXEC SQL select ime, adres, kurs
         into :ime, :adres :adres_ind, :kurs
         from student
         where fn = :fn;

    if (sqlca.sqlcode == 0) { /* SQLCODE instead sqlca.sqlcode */
        printf("Name   : %s\n",ime);
        if (adres_ind < 0)
           printf("Address: NULL\n");
        else
           printf("Address: %s\n",adres);
        printf("Course : %d\n", kurs);
    }
    else
        if (sqlca.sqlcode == 100 )
            printf("No student %s.\n", fn);
        else
            printf("SQL error %d\n", sqlca.sqlcode);

    EXEC SQL disconnect current;
    printf("\n DEMO1 Sample Program over.\n\n");
}
```

Пример. Програмата извлича факултетен номер, име и курс на студентите от специалност приложна математика и ги извежда на екрана (използва курсор).

```
#include <stdio.h>
EXEC SQL define NAME_LEN 30;

main()
{
EXEC SQL BEGIN DECLARE SECTION;
    char  ime[ NAME_LEN + 1 ];
    char  fn[7];
    short kurs;
```

```
EXEC SQL END DECLARE SECTION;

    printf( "       List of students in PM\n");
    printf( "FN     NAME                            COURSE\n");
    EXEC SQL connect to 'dbstudent';

    EXEC SQL declare stcursor cursor for
         select fn, ime, kurs
         from student
         where spec = 'PM'
         order by fn;
    EXEC SQL open stcursor;
    for (;;)
    {
      EXEC SQL fetch stcursor into :fn, :ime, :kurs;
      if (strncmp(SQLSTATE, "00", 2) != 0)
          break;
      printf("%s %s %d\n",fn, ime, kurs);
    }
    if (strncmp(SQLSTATE, "02", 2) != 0)
        printf("SQLSTATE after fetch is %s\n", SQLSTATE);
    else {
      EXEC SQL select count(*) into :kurs
                from student
                where spec='PM';
      printf("\n Count of students: %d\n", kurs);
    }
    EXEC SQL close stcursor;
    EXEC SQL free stcursor;

    EXEC SQL disconnect current;
    printf("\nDEMO2 Sample Program over.\n\n");
}
```

Пример. Програмата чете факултетен номер, извлича всички оценки на студента от таблица s_p, извежда ги на екрана по една и дава възможност на потребителя да промени оценката или да изтрие реда от s_p (използва курсор за обновяване).

```
#include <stdio.h>
EXEC SQL define NAME_LEN 30;

main()
{
EXEC SQL BEGIN DECLARE SECTION;
    char fn[7];
    char ime[NAME_LEN + 1];
    float ocenka;
    short ocenka_ind;
EXEC SQL END DECLARE SECTION;
char answer[10];

    printf( "DEMO3 Sample ESQL Program running.\n\n");
    EXEC SQL connect to 'dbstudent';
    EXEC SQL declare stupd cursor for
            select fn, ime_p, ocenka
            from s_p
            where fn = :fn
            for update of ocenka;

    printf("Enter FN: ");
    scanf("%s",fn);

    EXEC SQL whenever sqlerror goto err;
```

```
    EXEC SQL whenever not found goto done;

    EXEC SQL open stupd;
    for (;;)
      {
      EXEC SQL fetch stupd into :ime, :ocenka :ocenka_ind;
      if (ocenka_ind < 0)
            printf("%s\n",ime);
      else
            printf("%s %11.2f\n",ime, ocenka);
      printf("Next/Delete/Update/Quit: ");
      scanf("%s",answer);
      switch (answer[0]) {
        case 'N':
          break;
        case 'D':
          EXEC SQL delete from s_p where current of stupd;
          break;
        case 'U':
          printf("Enter new ocenka: ");
          scanf("%f", &ocenka);
          EXEC SQL update s_p set ocenka=:ocenka where current of stupd;
          break;
        case 'Q':
          goto done;
        }
      }

done:
    EXEC SQL close stupd;
    EXEC SQL free stupd;

    EXEC SQL disconnect current;
    printf("\nDEMO3 Sample Program over.\n\n");
    exit(0);

err:
    printf("SQL error: SQLSTATE is %s\n", SQLSTATE);
    exit(1);
}
```

Следват няколко примера написани на Informix - 4GL за БД-студенти.

Пример. Програмата чете факултетен номер, извлича името, адреса и курса на студента от БД и ги извежда на екрана.

```
DATABASE dbstudent
MAIN
      DEFINE name LIKE student.ime,
             fnom LIKE student.fn,
             adr LIKE student.adres,
             cours LIKE student.kurs
      PROMPT "Enter a student FN: " FOR fnom
      SELECT fn, ime, adres, kurs INTO fnom, name, adr,cours
      FROM student WHERE fn = fnom
      IF status = NOTFOUND THEN
            DISPLAY " No student with this FN."
      ELSE
            DISPLAY "FN     : ", fnom, "  ", name
            DISPLAY "Address: ", adr
            DISPLAY "Course : ", cours
      END IF
END MAIN
```

Пример. Програмата чете номер на курс и извлича извлича факултетен номер, име и група на студентите от сътветния курс и ги извежда на екрана (използва курсор).

```
DATABASE dbstudent
MAIN
      DEFINE name LIKE student.ime,
             fnom LIKE student.fn,
             cours LIKE student.kurs,
             group LIKE student.grupa
      PROMPT "Enter a course: " FOR cours
      DECLARE stcurs CURSOR FOR
            SELECT fn, ime[1,20], grupa
            FROM student WHERE kurs = cours
            ORDER BY grupa, fn
      DISPLAY "FN     Name                 Group "
      DISPLAY "---------------------------------"
      OPEN stcurs
      WHILE TRUE
            FETCH stcurs INTO fnom, name, group
            IF status != 0 THEN
                  EXIT WHILE
            END IF
            DISPLAY  fnom,"  ", name, group
      END WHILE
      IF status = NOTFOUND THEN
            SELECT COUNT(*) INTO group
            FROM student
            WHERE kurs = cours
            DISPLAY "Count of students: "
      END IF
      CLOSE stcurs
END MAIN
```