SQL language: advanced constructs

SQL for applications

DBMG

SQL for applications

- Introduction
- Cursors
- Updatability
- Static and dynamic SQL
- Embedded SQL
- Call Level Interface (CLI)
- Stored Procedures
- Comparison of altrernatives

DBMG

SQL per applications

Introduction

DBMG

Example application

- Banking operations
  - withdrawal operation from an account through an ATM
  - withdrawal operation from an account at a bank counter

Withdrawal from an ATM

- Operations performed
  - check the validity of ATM card and PIN code
  - select withdrawal operation
  - specify the required amount
  - verify availability
  - store the operation
  - update the account balance
  - dispense the required amount of money

Withdrawal from an ATM

- Access to a database is required to carry out many of the listed operations
  - by executing SQL commands
- The operations must be executed in an appropriate order

Withdrawal at a bank counter

- Operations performed
  - verify the identity of the customer
  - communicate the intention to withdraw money
  - verify availability
  - store the operation
  - update the account balance
  - dispense the required a mount of money

Withdrawal at a bank counter

- Access to a database is required to carry out many of the listed operations
  - By executing SQL commands
- The operations must be executed in an appropriate order

Example: banking operations

- Banking operations require accessing the database and modifying its contents
  - execution of SQL commands
    - customers or the bank employees are not directly executing the SQL commands
  - an application hides the execution of the SQL commands
- Correctly managing banking operations requires executing a specific sequence of steps
  - an application allows specifying the correct order of execution for the operations

Applications and SQL

- Real problems can hardly ever be solved by executing single SQL commands
- We need applications to
  - acquire and handle input data
    - user choices, parameters
  - manage the application logic
    - flow of operation to execute
  - Return results to the user using different formats
    - on-relational data representation
      - XML document
    - complex data visualization
      - graphs, reports

Integrating SQL and applications

- Applications are written in traditional high-level programming languages
  - C, C++, Java, C#, ...
  - the language is called *host language*
- SQL commands are used in the applications to access the database
  - queries
  - updates

Integrating SQL and applications

- It is necessary to integrate the SQL language with programming languages
  - SQL
    - declarative language
  - programming languages
    - usually procedural

Impedance mismatch

- Impedance mismatch
  - SQL queries operate on one or more tables and produce a table as a result
    - set-oriented approach
  - Programming languages access tables by reading rows *one by one*
    - tuple-oriented approach
- Possible solutions to solve the conflict
  - use cursors
  - Use languages that intrinsically provide data structures storing «set of rows»

SQL and programming languages

- Main integration tecniques
  - Embedded SQL
  - Call Level Interface (CLI)
    - SQL/CLI, ODBC, JDBC, OLE DB, ADO.NET, ..
  - Stored procedures
- Classified as
  - client-side
    - embedded SQL, call level interface
  - server-side
    - stored procedures

Client-side approach

- The application
  - is outside the DBMS
  - contains all of the application logic
  - requires that the DBMS execute SQL commands and return the result
  - processes the data returned by the DBMS

Server-side approach

- The application (or part of it)
  - is inside the DBMS
  - all or part of the application logic is moved inside the DBMS

Client-side approach vs. server-side approach

- Client-side approach
  - greater indipendence from the DBMS employeed
  - lower efficiency
- Server-side approach
  - depends on the DBMS employed
  - higher efficiency

SQL for applications

Cursors

DBMG

Impedance mismatch

- Main problem in the integration between SQL and programming languages
  - SQL queries operate on one or more tables and produce a table as a result
    - set-oriented approach
  - programming languages access tables by reading rows *one by one*
    - tuple-oriented approach

DBMG

Cursors

- If an SQL command returns a single row
  - it is sufficient to specify in which host language variable the result of the command shall be stored
- If an SQL command returns a table (i.e., a set of tuple)
  - a method is required to read one tuple at a time from the query result (and pass it to the program)
    - Use of a *cursor*

Supplier and product DB

P

PId	PName	Color	Size	Store
P1	Jumper	Red	40	London
P2	Jeans	Green	48	Paris
P3	Blouse	Blue	48	Rome
P4	Blouse	Blue	44	London
P5	Skirt	Blue	40	Paris
P6	Shorts	Red	42	London

SP

SId	PId	Qty
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P3	200
S4	P4	300
S4	P5	400

S

SId	SName	#Employees	City
S1	Smith	2	London
S2	Jones	1	Paris
S3	Blake	3	Paris
S4	Clark	2	London
S5	Adams	3	Athens

Example no.1

- Show the name and the number of employees for the supplier with code S1

```
SELECT SName, #Employees
FROM   S
WHERE SId='S1';
```

- The query returns *at most* one tuple

SName	#Employees
Smith	2

- It is sufficient to specify in which host language variables the selected tuple must be stored

Example no.2

- Show the name and the number of employees of the suppliers based in London

```
SELECT SName, #Employees
FROM   S
WHERE City='London';
```

- The query returns a set of tuples

SName	#Employees
Smith	2
Clark	2

Cursor

- It is necessary to define a *cursor* to read each tuple from the result separately

Example no. 2

- Definition of a cursor with the Oracle PL/SQL syntax

```
CURSOR LondonSuppliers IS
SELECT SName, #Employees
FROM   S
WHERE City=‘London’;
```

Cursors

- A cursor allows reading the individual tuples from the result of a query
  - It must be associated with a specific query
- Each SQL query that may return a set of tuples *must be associated with* a cursor

DBMG

Cursors

- Cursors are not required
  - for SQL queries that may return at most one tuple
    - selections on the primary key
    - aggregation operations without a GROUP BY clause
  - for update and DDL commands
    - they don't generate any tuple as a result

SQL for applications

Updatability

DBMG

Updatability

- The tuple currently pointed to by the cursor may be updated or deleted
  - more efficient than executing a separate SQL update command
- Updating a tuple with a cursor is possible only if the view that corresponds to the associated query may be updated
  - there must exist a one-to-one correspondance between the tuple pointed to by the cursor and the tuple to update in the database table

Example: non-updatable cursor

- Let us consider the *SupplierData* cursor associated with the following query:

```
SELECT DISTINCT SId, SName,
#Employees
FROM   S, SP, P
WHERE S.SId=SP.SId
         AND P.PId=SP.PId
         AND Color=‘Red’;
```

- The *SupplierData* cursor is *not* updatable
- By rewording the query, the cursor becomes updatable

Example: updatable cursor

- Let us suppose the *SupplierData* cursor is now associated with the following query:

```
SELECT SId, SName, #Employees
FROM   S
WHERE SId IN (SELECT SId
              FROM   SP, P
              WHERE SP.PId=P.PId
              AND Color='Red');
```

- The two queries are equivalent
  - the result of the new query is the same
- The *SupplierData* cursor is updatable

SQL for applications

Static and dynamic SQL

DBMG

Static SQL

- SQL commands to execute are known during the application writing
  - the definition of each SQL command is known
  - commands can contain variables
    - The value of the variables is known only during the execution of the SQL command

Static SQL

- The definition of the SQL commands takes place during the writing of the application
  - simplifies the application writing
    - The structure of queries and results is known a priori
  - makes the a priori optimization of the SQL commands possible
    - during the application compiling phase, the DBMS optimizer
      - compiles the SQL command
      - creates the execution plan
    - such operations are no longer necessary during the execution of the application
      - more efficient execution

Dynamic SQL

- The SQL commands to follow *are not* known during the writing of the application
  - the SQL coomands are dynamically defined by the application in the execution phase
    - they depend on the executed applicative flow
  - the SQL commands can be provided by the user as input

Dynamic SQL

- The definiton at execution time of the SQL commands
  - allows to define more complex applications
    - offers a major flexibility
  - makes the application writing harder
    - the format of the query result is not known during the writing
  - makes the execution less efficient
    - durante each application execution, it is necessary to compile and optimize every SQL command

Dynamic SQL

- It is possible to reduce the execution time if the same dynamic query has to be executed more than once in *in the same work session*
  - the compilation and the choice of the execution plan are carried out only once
  - the query is executed more than once (with different values of the variables)

DBMG

SQL for applications

Embedded SQL

DBMG

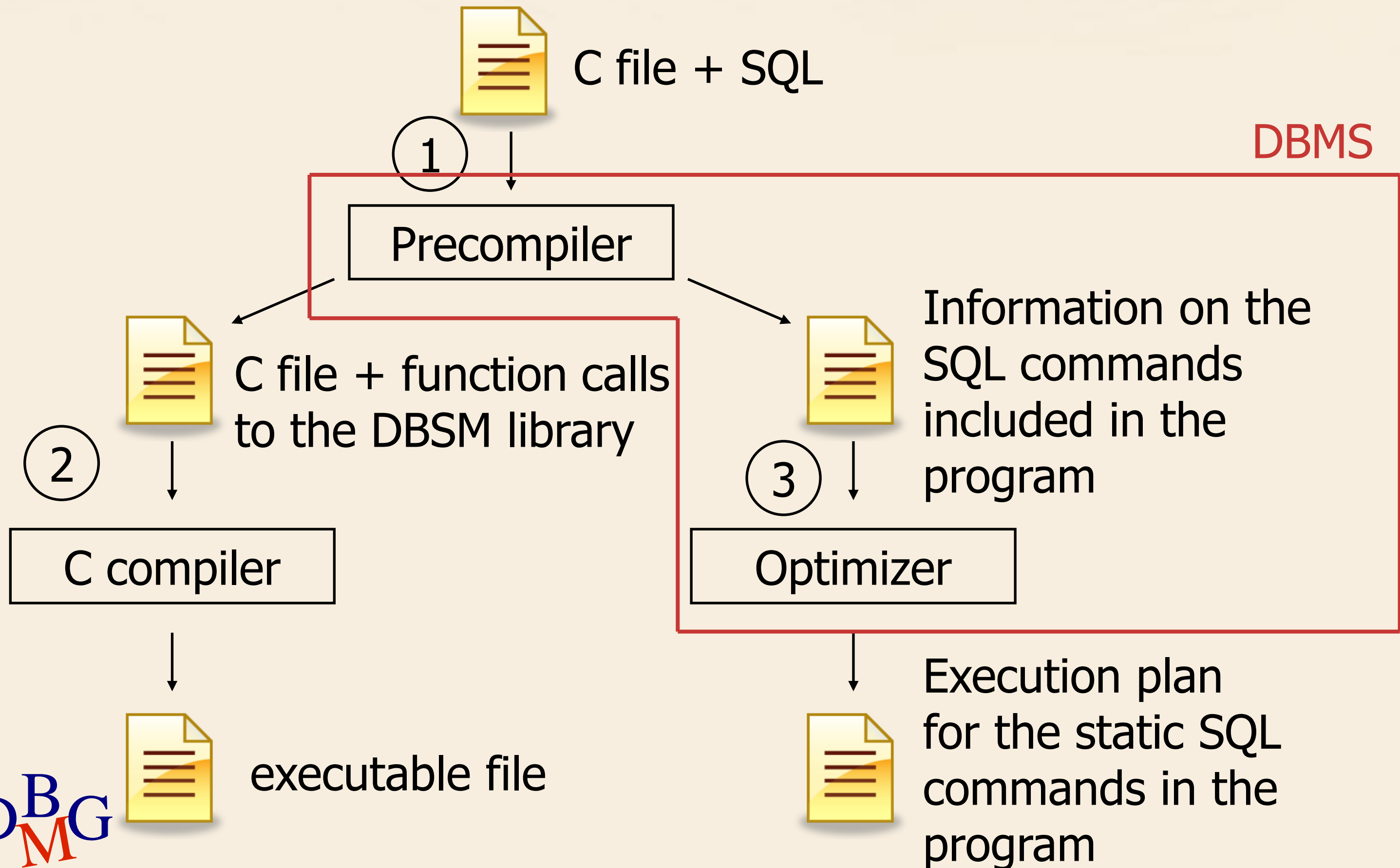
Embedded SQL

- SQL commands are «embedded» in the application weitten in a traditional programming language (C, C++, Java, ..)
  - the SQL syntax is different from that of the host language
- SQL commands cannot be directly compiled by a normal compiler
  - they must be recognized
    - they are preceded by the EXEC SQL keyword
  - they must be replaced with appropriate commands in the host programming language

Precompilation

- The precompiler
  - identifies SQL commands embedded in the code
    - parts preceded by EXEC SQL
  - replaces the SQL commands with function calls to specific APIs of the chosen DBMS
    - such functions are written in the host programminh languafe
  - it optionally sends the static SQL commands to the DBMS for compilation and optimization
- the precompiler is tied to a specific DBMS

Embedded SQL: compilation



Precompiler

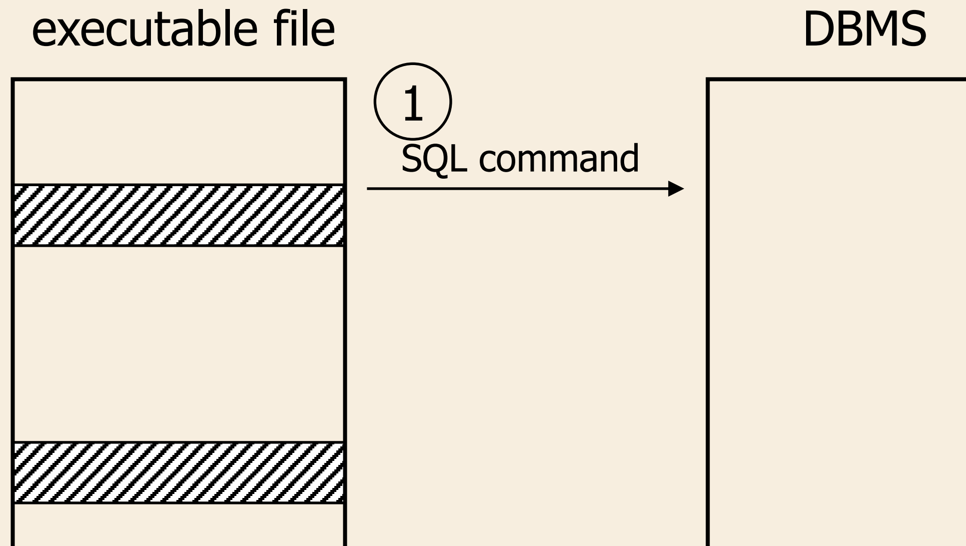
- The precompiler depends on three elements of the system architecture
  - host language
  - DBMS
  - operating system
- The appropriate compiler for the architecture choice must be employed

Embedded SQL: execution

- During the program execution
  1. During the program execution
     - it calls a DBMS library function

DBMG

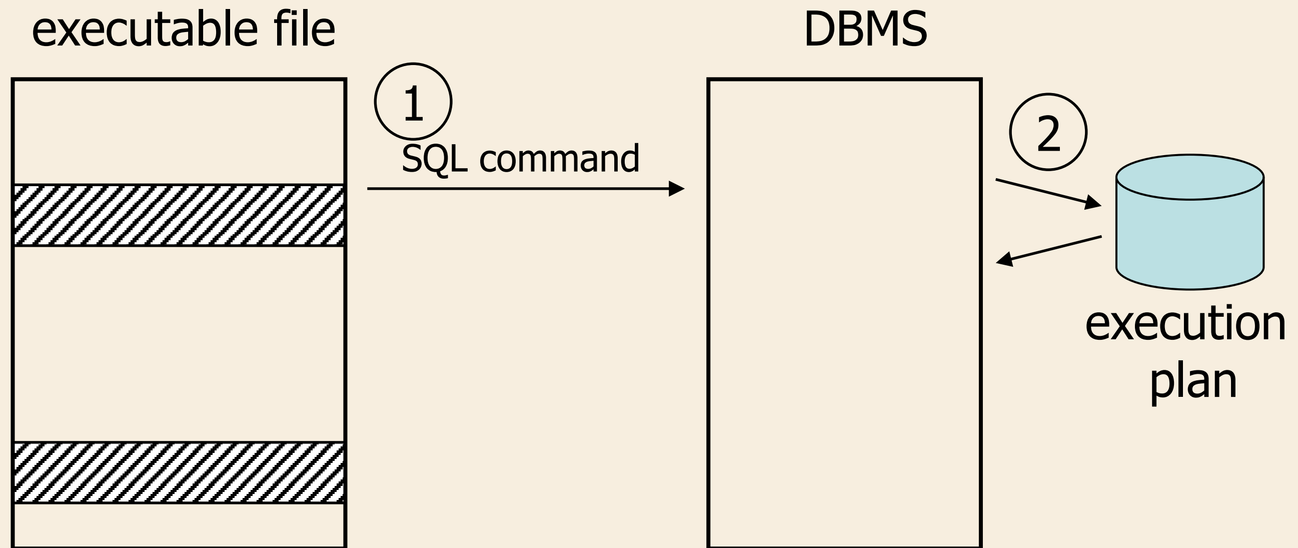
Embedded SQL: execution



Embedded SQL: execution

- During the program execution
  1. The program sends an SQL command to the DBMS
     - it calls a DBMS library function
  2. The DBMS generates the execution plan for the command
     - if one has already been defined, it will be retrieved

Embedded SQL: execution

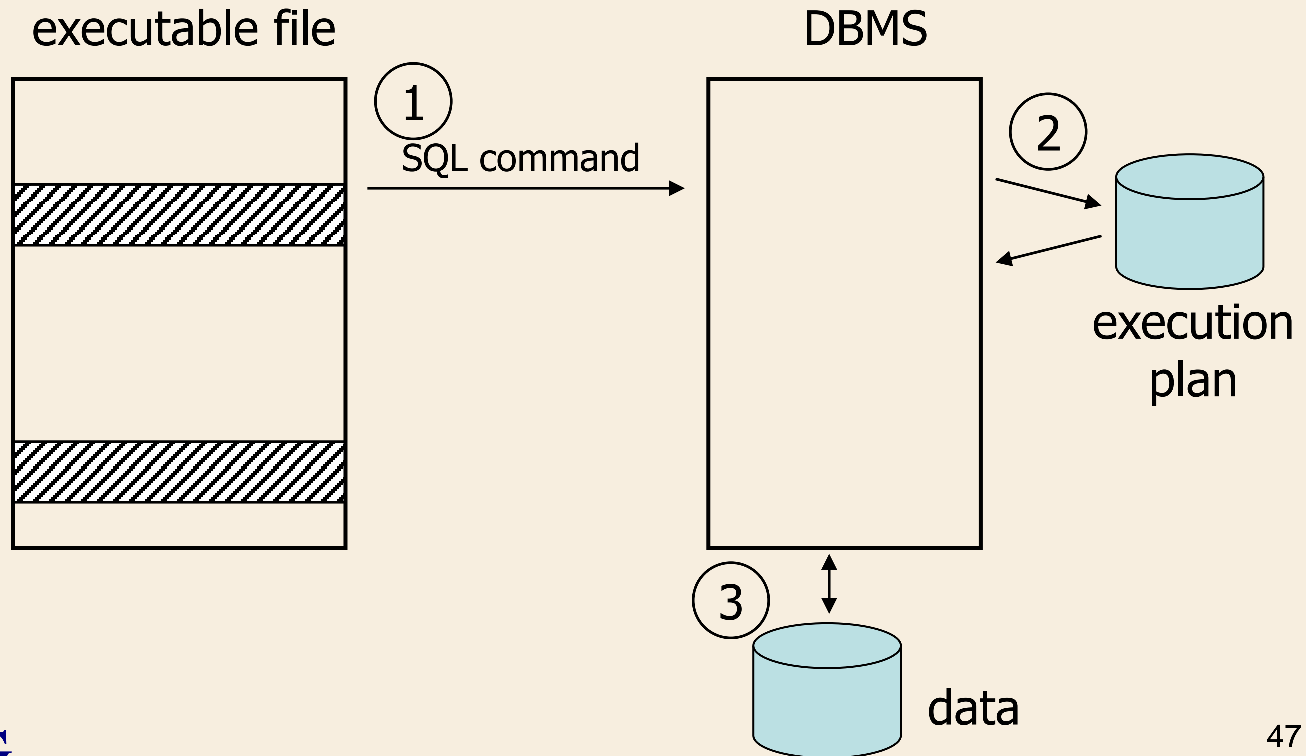


Embedded SQL: execution

- During the program execution
  1. The program sends an SQL command to the DBMS
     - it calls a DBMS library function
  2. The DBMS generates the execution plan for the command
     - if one has already been defined, it will be retrieved
  3. The DBMS executes the SQL command

DBMG

Embedded SQL: execution

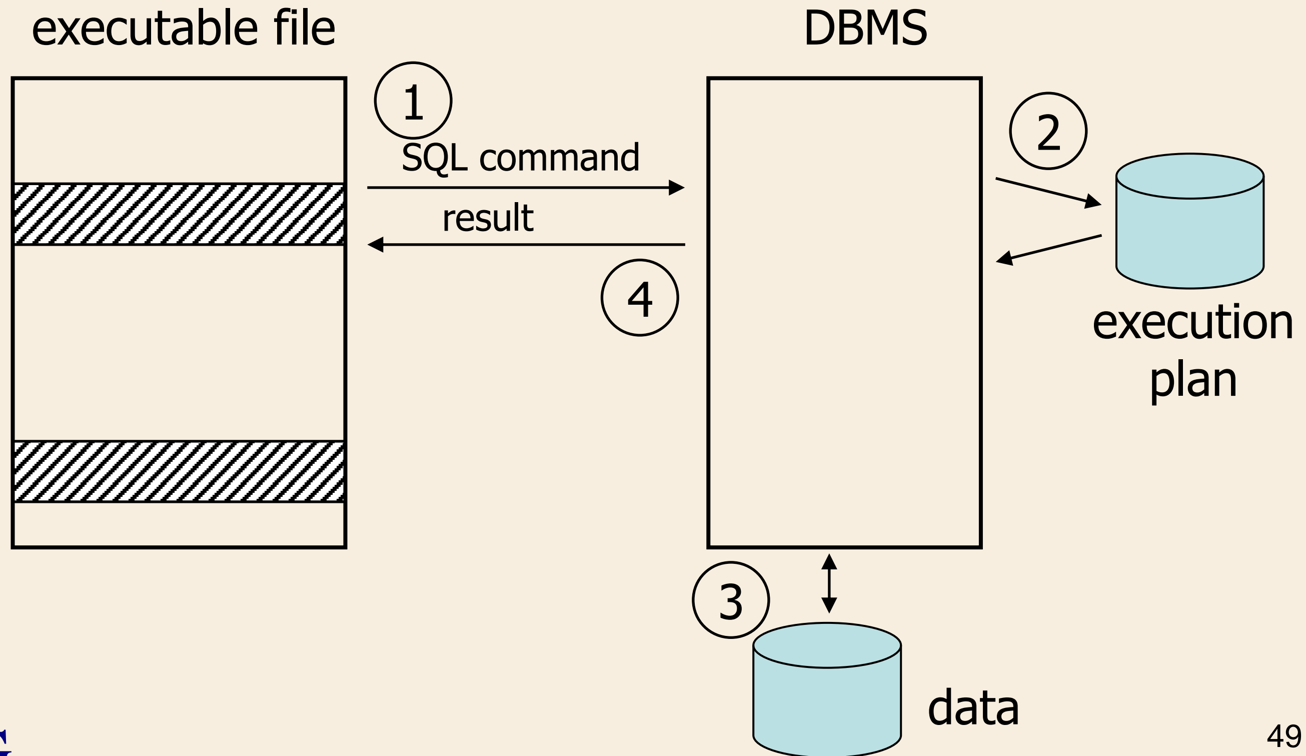


Embedded SQL: execution

- During the program execution
  1. The program sends an SQL command to the DBMS
     - it calls a DBMS library function
  2. The DBMS generates the execution plan for the command
     - if one has already been defined, it will be retrieved
  3. The DBMS executes the SQL command
  4. The DBMS returns the result of the SQL command
     - a transit area is used as temporary data storage

DBMG

Embedded SQL: execution



Embedded SQL: execution

- During the program execution
  1. The program sends an SQL command to the DBMS
     - it calls a DBMS library function
  2. The DBMS generates the execution plan for the command
     - if one has already been defined, it will be retrieved
  3. The DBMS executes the SQL command
  4. The DBMS returns the result of the SQL command
     - a transit area is used as temporary data storage
  5. The programm processes the result

Example of embedded SQL code

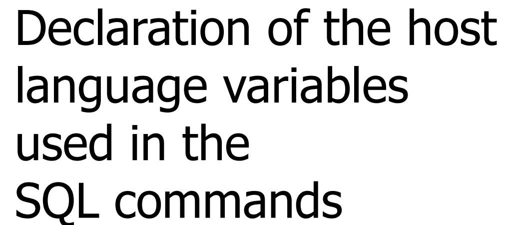
```
#include <stdlib.h>
.....
EXEC SQL BEGIN DECLARE SECTION
char VarSId[6];
int NumEmployees;
char City[16];
EXEC SQL END DECLARE SECTION

int alpha, beta;
....
EXEC SQL DECLARE S TABLE (SId CHAR(5) NOT NULL,
                          SName CHAR(20) NOT NULL,
                          NumEmployees SMALLINT NOT NULL,
                          City CHAR(15) NOT NULL);
.....
```

Example of embedded SQL code

```
EXEC SQL INCLUDE SQLCA;
.....
if (alpha>beta) {
  EXEC SQL SELECT NumEmployees, City
                  INTO :NumEmployees, :City
                  FROM S
                  WHERE SId=:VarSId;


  printf("%d %s", NumEmployees, City);
  ......
}
.....
```

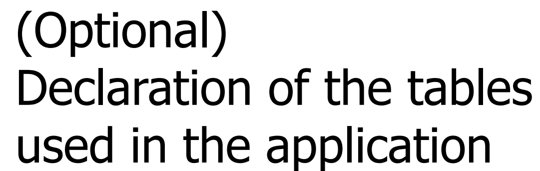
Example of embedded SQL code

```
#include <stdlib.h>
.....
EXEC SQL BEGIN DECLARE SECTION
char VarSId[6];
int NumEmployees;
char City[16];
EXEC SQL END DECLARE SECTION

int alpha, beta;
....
EXEC SQL DECLARE S TABLE (SId CHAR(5) NOT NULL,
                          SName CHAR(20) NOT NULL,
                          NumEmployees SMALLINT NOT NULL,
                          City CHAR(15) NOT NULL);
.....
```



Example of embedded SQL code

```
#include <stdlib.h>
.....
EXEC SQL BEGIN DECLARE SECTION
char VarSId[6];
int NumEmployees;
char City[16];
EXEC SQL END DECLARE SECTION

int alpha, beta;
....
EXEC SQL DECLARE S TABLE (SId CHAR(5) NOT NULL,
                          SName CHAR(20) NOT NULL,
                          NumEmployees SMALLINT NOT NULL,
                          City CHAR(15) NOT NULL);
.....
```



Example of embedded SQL code

Declaration of the communication area

```
EXEC SQL INCLUDE SQLCA;
.....
if (alpha>beta) {
  EXEC SQL SELECT NumEmployees, City
                  INTO :NumEmployees, :City
                  FROM S
                  WHERE SId=:VarSId;


  printf("%d %s", NumEmployees, City);
  ......
}
.....
```

Example of embedded SQL code

```
EXEC SQL INCLUDE SQLCA;
.....
if (alpha>beta) {
  EXEC SQL SELECT NumEmployees, City
                INTO :NumEmployees, :City
                FROM S
                WHERE SId=:VarSId;

  printf("%d %s", NumEmployees, City);
  ......
}
.....
```

Example of embedded SQL code

```
EXEC SQL INCLUDE SQLCA;
.....
if (alpha>beta) {
  EXEC SQL SELECT NumEmployees, City
                  INTO :NumEmployees, :City
                  FROM S
                  WHERE SId=:VarSId;

  printf("%d %s", NumEmployees, City);
  ......
}
.....
```

Variables of the host language

- It is possible to introduce in the SQL commands references to variables of the host language
  - variables in reading
    - allow the interactive execution of the commands
    - the variables are used as parameters in the selection predicates instead of the constants
  - variables in writing
    - variables in which the current tuple is stored
    - indicated after the keyword INTO in the commands SELECT and FETCH

Variables of the host language

- In the programs
  - the declaration of the variables is limited by the couple of commands
    - EXEC SQL BEGIN DECLARE SECTION
    - EXEC SQL END DECLARE SECTION
  - in the SQL commands the variables are preceeded by the symbol ":" in order to distinguish them from the names of the columns

DBMG

Type check

- The type of the variables must be compatible with the type of the corresponding SQL columns
  - the names of the variables and of the SQL columns can be the same

Semantic check

- Each SQL DML command must refer to objects that have already been defined into the database
- The precompiler carries out the semantic check of the SQL commands
  - accessing the database in order to find the schema of the referenced objects in the data dictionary
    - It is necessary to be able to connect to the DBMS during the precompilation of the code
  - or considering the definitions of the tables present in the code
    - EXEC SQL DECLARE command

Transit area

- A communication area between the DBMS and the host language must be defined
  - some precompilers automatically include the definition of the communication area
  - in other cases it is necessary to use the command

```
EXEC SQL INCLUDE SQLCA
```

- It is necessary to have apposite variables to store the status of the last SQL command executed
  - variable SQLCA.SQLCODE
    - automatically defined

Execution of SQL commands

- Embedded SQL allow to execute all kinds of SQL commands
  - DML
  - DDL
- Execution of an SQL command

EXEC SQL *SQLCommand*;

Execution of SQL commands

- After the execution it is possible to checkthe status of the executed command with the variable SQLCA.SQLCODE
  - command correctly executed

    SQLCODE=0
  - command not executed because of an error

    SQLCODE≠0
    - the value of SQLCODE specifies the type of error

Update commands and DDL

- Command that do not return a set of tuple
  - it is necessary to check if the operation ended properly

    SQLCODE=0
  - there are no results to analyze
    - the use of cursors is not necessary

- It works differently depending on the number of tuples returned by the query
  - a single tuple
    - execution of the SELECT command
    - indication of the variables where the result must be stored directly in the SELECT command
      - the use of cursors is not necessary
  - a set of tuples
    - definition and use of a *cursor* associated to the SELECT command
    - indication of the variables where the single tuples read in the FETCH command must be stored

Example: selection of a single tuple

- Select the number of emplotees and the city of the supplier whose code value is contained in the host variable *VarSId*

```
EXEC SQL SELECT #Employees, City INTO
    :NumEmployees, :City
            FROM S
            WHERE SId = :VarSId;
```

- In the SQL query the variables where the results must be stores are declared after the keyword INTO

Example: selection of a single tuple

- Select the number of emplotees and the city of the supplier whose code value is contained in the host variable *VarSId*

```
EXEC SQL SELECT #Employees, City INTO
    :NumEmployees, :City
            FROM S
            WHERE SId = :VarSId;
```

- In the WHERE it is possible to use variables of the host language insted of constants

Example: selection of a single tuple

- The status of the operation must be always checked after the end of the operation
  - SQLCODE = 0
    - query properly executed
    - the selected values have been stored in the variables indicated in the query (NumEmployees and City)
  - SQLCODE = 100
    - no tuple satisfies the predicate
  - SQLCODE < 0
    - execution error
      - more than a record satisfies the predicate
      - table not available
      - …

DBMG

Cursor

- The cursor allows to read individually the tuples that belong to the result of a query
  - it must be associated to a specific query
- Each SQL query that can return a set of tuples *must be associated* to a cursor

Operations on the cursors

- Basic operations on the cursors
  - declaration
  - opening
  - reading (tipically inside a cycle)
  - closing
- Similar to the management modes of a file

Declaration

- DECLARE command
  - declaration of the cursor structure
    - assignement of a name to the cursor
    - definition of the query associated to the cursor

DECLARE command

EXEC SQL DECLARE *CursorName* [SCROLL] CURSOR
FOR *SQLQuery*
[FOR <READ ONLY| UPDATE [OF *AttributesList*]>];

- READ ONLY option
  - the cursor can be used only for the reading of the result
  - default option

DECLARE command

```
EXEC SQL DECLARE CursorName [SCROLL] CURSOR
        FOR SQLQuery
        [FOR <READ ONLY| UPDATE [OF AttributesList]>];
```

- SCROLL option
  - the application can move freely on the result
    - reading forwards and backwards

DECLARE command

```
EXEC SQL DECLARE CursorName [SCROLL] CURSOR
        FOR SQLQuery
       [FOR <READ ONLY| UPDATE [OF AttributesList]>];
```

- UPDATE option
  - the cursor can be used in an update command
    - it is possible to specify which attributes shall be updated

Opening

- OPEN command
  - opening of the cursor
    - execution of the query on the database
    - memorization of threresult in a temporary area

EXEC SQL OPEN *CursorName*;

- After the opening the cursor can be found in the first tuple of the result

Reading

- FETCH command
  - reading of the next available tuple
    - memoriazation of the tuple in a variable of the host program
  - update of the cursor position
    - the cursor moves one line forward
      - to the next tuple
- It is necessary to define a cycle to read all the tuples of the result
  - the host language is used
  - every call to the FETCH command inside the cycle select a single tuple

FETCH command

```
EXEC SQL FETCH [Position FROM] CursorName
            INTO VariablesList;
```

- If the SCROLL option is present in the definition of the cursor, the *Position* parameter can assume the values
  - next, prior, first, last, absolute, relative
- Otherwise, it can only assume the value next
  - default value

Position of the cursor (1/2)

- *Position* values
  - next
    - reading of the line following the current one
  - prior
    - reading of the line preceding the current one
  - first
    - reading of the first line of the result
  - last
    - reading of the last line of the result

Position of the cursor (2/2)

- absolute *fullExpression*
  - reading of the *i* -th line of the result
    - the *i* position is the result of the full expression
- relative *fullExpression*
  - like absolute but the landmark is the current position

Closing

- CLOSE command
  - closing of the cursor
    - release of the temporary area containing the result of the query
      - the result of the query is no longer accessible
    - update of the database in case of cursors associated to updatable queries

EXEC SQL CLOSE *CursorName*;

Observations

- In an application, a cursor
  - is defined only once
  - can be used more than once
    - must be open and closed every time
- More cursors can be defined in the same application

DBMG

Example: selecting suppliers

- Show the code and the number of employees of the suppliers whose city is contained in the host variable *VarCity*
  - the value of *VarCity* is given by the user as a parameter of the application

Example: selecting suppliers

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/************************* Errors management *******************************/
void sql_error(char *msg)
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    fprintf(stderr,"\n%s\n", msg);
    fprintf(stderr,"Internal error code: %ld\n", sqlca.sqlcode);
    fprintf(stderr,"%s\n",sqlca.sqlerrm.sqlerrmc);
    EXEC SQL ROLLBACK;
    exit(EXIT_FAILURE);
}
```

Example: selecting suppliers

```
/********************************   MAIN   ***********************************/
int main(int argc,char **argv)
{
    EXEC SQL BEGIN DECLARE SECTION;
        char username[20]="user123";
        char password[20]=“pwd123";
        char VarCity[16];
        char SId[6];
        int  #Employees;
    EXEC SQL END DECLARE SECTION;

     /* Direct error management */
    EXEC SQL WHENEVER SQLERROR CONTINUE;

    /* Connection opening */
    EXEC SQL CONNECT TO supplies@127.0.0.1 USER :username IDENTIFIED BY :password;

    if (sqlca.sqlcode!=0)
        sql_error("Error in the connection phase");
```

Example: selecting suppliers

```
/* Cursor declaration */
EXEC SQL DECLARE selectedSuppliers CURSOR FOR
SELECT SId,#Employees FROM S WHERE City = :VarCity;

/* Setting the value of VarCity */
strcpy(VarCity,argv[1]);

/* Cursor opening */
EXEC SQL OPEN selectedSuppliers;

if (sqlca.sqlcode!=0)
    sql_error("Error in the cursor opening phase");

/* Print of the selected data */
printf("Suppliers list\n");
```

Example: selecting suppliers

```
do {
        EXEC SQL FETCH selectedSuppliers INTO :SId, :#Employees;
        /* Check the status of the last fetch operation */
        switch(sqlca.sqlcode)   {
                case 0: /* New tuple correctly read */
                { /* Print of the tuple */
                    printf("%s,%d",SId, #Employees);
                }
                break;

                case 100: /* No more data */
                break;

                default: /* Error */
                sql_error("Error in the data reading phase");
                break;
        }
}
while (sqlca.sqlcode==0);
```

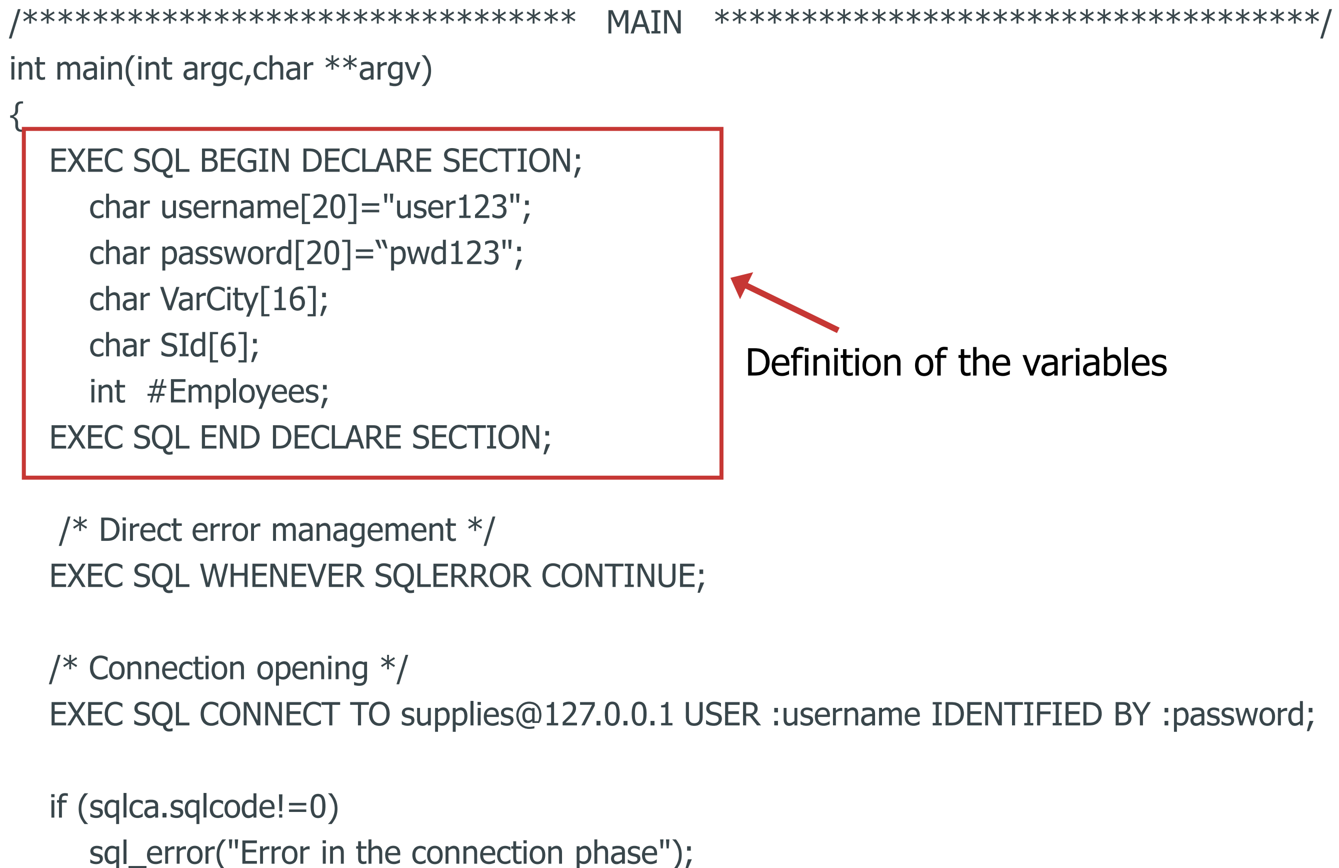
Example: selecting suppliers

```
    /* Cursor closing */
    EXEC SQL CLOSE selectedSuppliers;
}
```

Example: selecting suppliers

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/************************* Errors management ******************************/
void sql_error(char *msg)
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    fprintf(stderr,"\n%s\n", msg);
    fprintf(stderr,"Internal error code: %ld\n", sqlca.sqlcode);
    fprintf(stderr,"%s\n",sqlca.sqlerrm.sqlerrmc);
    EXEC SQL ROLLBACK;
    exit(EXIT_FAILURE);
}
```

Example: selecting suppliers



```
/********************************   MAIN   **********************************/
int main(int argc,char **argv)
{
  EXEC SQL BEGIN DECLARE SECTION;
     char username[20]="user123";
     char password[20]=“pwd123";
     char VarCity[16];
     char SId[6];
     int  #Employees;
  EXEC SQL END DECLARE SECTION;

   /* Direct error management */
  EXEC SQL WHENEVER SQLERROR CONTINUE;

  /* Connection opening */
  EXEC SQL CONNECT TO supplies@127.0.0.1 USER :username IDENTIFIED BY :password;

  if (sqlca.sqlcode!=0)
     sql_error("Error in the connection phase");
```

Example: selecting suppliers

```
/*********************************   MAIN   **********************************/
int main(int argc,char **argv)
{
   EXEC SQL BEGIN DECLARE SECTION;
      char username[20]="user123";
      char password[20]="pwd123";
      char VarCity[16];
      char SId[6];
      int  #Employees;
   EXEC SQL END DECLARE SECTION;

    /* Direct error management */
   EXEC SQL WHENEVER SQLERROR CONTINUE;

   /* Connection opening */
   EXEC SQL CONNECT TO supplies@127.0.0.1 USER :username IDENTIFIED BY :password;

   if (sqlca.sqlcode!=0)
      sql_error("Error in the connection phase");
```

Connection to the DBMS

Example: selecting suppliers

```
/* Cursor declaration */
EXEC SQL DECLARE selectedSuppliers CURSOR FOR
SELECT SId,#Employees FROM S WHERE City = :VarCity;

/* Setting the value of VarCity */
strcpy(VarCity,argv[1]);

/* Cursor opening */
EXEC SQL OPEN selectedSuppliers;

if (sqlca.sqlcode!=0)
    sql_error("Error in the cursor opening phase");

/* Print of the selected data */
printf("Suppliers list\n");
```

Example: selecting suppliers

```
/* Cursor declaration */
EXEC SQL DECLARE selectedSuppliers CURSOR FOR
SELECT SId,#Employees FROM S WHERE City = :VarCity;

/* Setting the value of VarCity */
strcpy(VarCity,argv[1]);

/* Cursor opening */
EXEC SQL OPEN selectedSuppliers;

if (sqlca.sqlcode!=0)
    sql_error("Error in the cursor opening phase");

/* Print of the selected data */
printf("Suppliers list\n");
```

Example: selecting suppliers

```
do {
        EXEC SQL FETCH selectedSuppliers INTO :SId, :#Employees;
        /* Check the status of the last fetch operation */
        switch(sqlca.sqlcode)   {
                case 0: /* New tuple correctly read */
                { /* Print of the tuple */
                    printf("%s,%d",SId, #Employees);
                }
                break;

                case 100: /* No more data */
                break;

                default: /* Error */
                sql_error("Error in the data reading phase");
                break;
        }
}
while (sqlca.sqlcode==0);
```

Example: selecting suppliers

```
do {
        EXEC SQL FETCH selectedSuppliers INTO :SId, :#Employees;
        /* Check the status of the last fetch operation */
        switch(sqlca.sqlcode)   {
                case 0: /* New tuple correctly read */
                { /* Print of the tuple */
                    printf("%s,%d",SId, #Employees);
                }
                break;

                case 100: /* No more data */
                break;

                default: /* Error */
                sql_error("Error in the data reading phase");
                break;
        }
}
while (sqlca.sqlcode==0);
```

Example: selecting suppliers

```
do {
        EXEC SQL FETCH selectedSuppliers INTO :SId, :#Employees;
        /* Check the status of the last fetch operation */
        switch(sqlca.sqlcode)   {
                case 0: /* New tuple correctly read */
                { /* Print of the tuple */
                    printf("%s,%d",SId, #Employees);
                }
                break;

                case 100: /* No more data */
                break;

                default: /* Error */
                sql_error("Error in the data reading phase");
                break;
        }
}
while (sqlca.sqlcode==0);
```

Example: selecting suppliers

```
/* Cursor closing */
EXEC SQL CLOSE selectedSuppliers;
}
```

Update with the cursors

- It is possible to update or delete the tuple pointed by a cursor
  - update

```
EXEC SQL UPDATE TableName
          SET ColumnName = Expression
              {, ColumnName = Expression}
          WHERE CURRENT OF CursorName;
```

  - delete

```
EXEC SQL DELETE FROM TableName
          WHERE CURRENT OF CursorName;
```

Update with the cursors

- The update and the deletion are possible if and only if
  - the cursor has been defined in an appropriate way
    - option FOR UPDATE in the command DECLARE
  - there is a one-to-one correspondance between the tuples of the result and the tuples present in the DBMS

DBMG

Transaction management

- In an embededd SQL program it is possible to define the limits of a transaction
  - begin of a transaction

    ```
    EXEC SQL BEGIN TRANSACTION;
    ```
  - succesful end of a transactiono

    ```
    EXEC SQL COMMIT;
    ```
  - failure of a transaction

    ```
    EXEC SQL ROLLBACK;
    ```

Transaction management

- Until the commands COMMIT or ROLLBACK are not explicitly invoked, the SQL operations of updating has to be considered “updating attempts”

SQL for applications

Call Level Interface (CLI)

DBMG

Call Level Interface

- Requests are sent to the DBMS by using ad-hoc functions of the host language
  - solution based on predefined interfaces
    - API, Application Programming Interface
  - the SQL commands are passed to the host language functions as parameters
  - there is no precompiler
- The host program directly includes calls to the functions provided by the API

Call Level Interface

- Different solutions are available using the Call Level Interface (CLI) paradigm
  - standard SQL/CLI
  - ODBC (Open DataBase Connectivity)
    - proprietary SQL/CLI solution by Microsoft
  - JDBC (Java Database Connectivity)
    - solution for the Java environment
  - OLE DB
  - ADO
  - ADO.NET

Usage pattern

- Regardless of the specific CLI solution adopted, the interaction with the DBMS has a common structure
  - open a connection to the DBMS
  - execute SQL commands
  - close the connection

Interaction with the DBMS

1. Call an API primitive to create a connection to the DBMS
2. Send an SQL command across the connection
3. Receive a result in response to the command
   - i.e., a set of tuples, in the case of a SELECT command
4. Process the result obtained from the DBMS
   - ad-hoc primitives allow reading the result
5. Close the connection at the end of the working session

JDBC (Java Database Connectivity)

- CLI solution for the JAVA environment
- The architecture comprises
  - a set of standard classes and interfaces
    - used by the Java programmer
    - indipendent of the DBMS
  - a set of “proprietary”classes (drivers)
    - implementing the standard classes and interfaces to provide communication with a specific DBMS
    - dependent on the DBMS
    - invoked at runtime
      - not required at the time when the application is compiled

DBMG

JDBC: interaction with the DBMS

- Load the specific driver for the DBMS of choice
- Create a connection
- Execute SQL commands
  - create a statement
  - submit the command for execution
  - process the result (in the case of queries)
- Close the statement
- Close the connection

Loading the DBMS driver

- The driver is specific to the DBMS employed
- It is loaded through dynamic instantiation of the class associated with the driver

```
Object Class.forName(String driverName)
```

  - driverName contains the name of the class to be instantiated
    - e.g., "oracle.jdbc.driver.OracleDriver"

Loading the DBMS driver

- It is the first operation to do
- We don’t need to know at compile time which DBMS we will be using
  - the name of the driver may be read at runtime from a configuration file

Creating a connection

- Invoke the getConnection method of the DriverManager class

  Connection DriverManager.getConnection(String url, String user, String password)

  - url
    - contains the information required to identify the DBMS to which we are connecting
    - the format depends on the specific driver
  - user and password
    - credentials for authentication

DBMG

Executing SQL commands

- The execution of an SQL command requires the use of a specific interface
  - called Statement
- Each Statement object
  - is associated with a connection
  - is created through the createStatement method of the Connection class

```
Statement createStatement()
```

DBMG

Update and DDL commands

- The execution of the command requires the following method on a Statement object

  int executeUpdate(String SQLCommand)

  - SQLCommand
    - the SQL command to be executed
  - the method returns
    - the number of processed (i.e., inserted, modified, deleted) tuples
    - a value of 0 for DDL commands

DBMG

Queries

- Immediate query execution
  - the server compiles and immediately executes the SQL command received
- "Prepared" query execution
  - useful when the same SQL command must be executed multiple times in the same working session
    - only the values of parameters may change
  - the SQL command
    - is compiled (prepared) only once and its execution plan is stored by the DBMS
    - is executed several times throughout the session

DBMG

Immediate execution

- It can be requested by invoking the following method on a Statement object

  ResultSet executeQuery(String SQLCommand)

  - SQLCommand
    - the SQL command to be executed
  - the method always returns a collection of tuples
    - an object of the ResultSet type
  - it handles in the same way queries that
    - return at most a single tuple
    - may return multiple tuples

Reading the result

- The ResultSet is analogous to a cursor
  - it provides methods to
    - move throughout the lines in the result
      - next()
      - first()
      - ...
    - extract the values of interest from the current tuple
      - getInt(String attributeName)
      - getString(String attributeName)
      - ....

Prepared statements

- A “prepared” SQL command is
  - compiled only once
    - at the beginning of the program execution
  - executed multiple times
    - the current values for the parameters must be specified before each execution
- A useful device when the execution of the same SQL command must be repeated several times
  - it reduces execution times
    - the compilation is done only once

DBMG

Preparing the Statement

- An object of the PreparedStatement type is used
  - created by means of the following method

    PreparedStatement prepareStatement(String SQLCommand)

    - SQLCommand
      - it contains the SQL command to be executed
      - the "?" symbol is used as a placeholder to indicate the presence of a parameter whose value must be specified
- Example

```
PreparedStatement pstmt;
pstmt=conn.preparStatement("SELECT SId, NEmployees
  FROM S WHERE City=?");
```

Setting parameters

- Replace "?" symbols for the current execution
- One of the following methods is invoked on a PreparedStatement object
  - void setInt(int parameterIndex, int value)
  - void setString(int parameterIndex, String value)
  - …
    - parameterIndex indicates the position of the parameter whose value is being assigned
      - the same SQL command may include several parameters
      - the index of the first parameter is 1
    - value indicates the value to be assigned to the parameter

DBMG

Execution of the prepared command

- An appropriate method is invoked on the PreparedStatement object
  - SQL query

    ResultSet executeQuery()
  - update

    ResultSet executeUpdate()
- The two methods have no input parameters
  - everything has been defined in advance
    - the SQL command to be executed
    - its execution parameters

Example: prepared statements

```
.....
PreparedStatement pstmt=conn.prepareStatement("UPDATE P
    SET Color=? WHERE PId=?");

/* Assign color Crimson to product P1 */
pstmt.setString(1,"Crimson");
pstmt.setString(2,"P1");
pstmt.executeUpdate();

/* Assign color SteelBlue to product P5 */
pstmt.setString(1,"SteelBlue");
pstmt.setString(2,"P5");
pstmt.executeUpdate();
```

Example: prepared statements

```
.....
PreparedStatement pstmt=conn.prepareStatement("UPDATE P
    SET Color=? WHERE PId=?");

/* Assign color Crimson to product P1 */
pstmt.setString(1,"Crimson");
pstmt.setString(2,"P1");
pstmt.executeUpdate();

/* Assign color Crimson to product P5 */
pstmt.setString(1,"SteelBlue");
pstmt.setString(2,"P5");
pstmt.executeUpdate();
```

Closing statement and connection

- As soon as a statement or a connection are no longer needed
  - they must be immediately closed
- Resources previously allocated to the statement or the connection can be released
  - by the application
  - by the DBMS

Closing a statement

- Closing a statement
  - is done by invoking the close method on a Statement object
    - void close()
- The resources associated with the corresponding SQL command are released

Closing a connection

- Closing a connection
  - is necessary when it is no longer required to interact with the DBMS
  - closes communication with the DBMS and releases the corresponding resources
    - also closes all statements associated with the connection
  - is done by invoking the close method on the Connection object
    - void close()

DBMG

Exceptions management

- Errors are handled through SQLException exceptions
- The SQLException contains
  - a string that describes the error
  - a string that identifies the exception
    - in a manner consistent with Open Group SQL Specification
  - an error code specific to the used DBSM

Example: selecting suppliers

- Print the codes and the number of employees of the suppliers whose city is stored in host variable *VarCity*
  - the value of *VarCity* is provided by the user as a parameter of the application

Example: selecting suppliers

```
import java.io.*;
import java.sql.*;

class CitySuppliers {

  static public void main(String argv[])  {
      Connection conn;
      Statement stmt;
      ResultSet rs;
      String query;
      String VarCity;

      /* Driver registration */
      try  {
          Class.forName("oracle.jdbc.driver.OracleDriver");
      }
      catch(Exception e)  {
          System.err.println("Driver unavailable: "+e);
      }
```

Example: selecting suppliers

```
try {
    /* Connection to the database */
   conn=DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe",
          “user123”,“pwd123");

   /* Creation of a statement for immediate commands */
   stmt = conn.createStatement();

   /* Assembling a query */
   VarCity =argv[0];
   query="SELECT SId, NEmployees FROM S WHERE City = '"+VarCity+"'";

   /* Execution of the query */
   rs=stmt.executeQuery(query);
```

Example: selecting suppliers

```
          System.out.println("Suppliers based in "+VarCity);
          /* Scan tuples in the result */
          while (rs.next()) {
             /* Print the current tuple */
              System.out.println(rs.getString("SId")+","+rs.getInt("NEmployees"));
          }
          /* Close resultset, statement and connection */
          rs.close();
          stmt.close();
          conn.close();
      }
      catch(Exception e) {
           System.err.println("Error: "+e);
      }
   }
}
```

Example: selecting suppliers

```
import java.io.*;
import java.sql.*;

class CitySuppliers {

  static public void main(String argv[])  {
      Connection conn;
      Statement stmt;
      ResultSet rs;
      String query;
      String VarCity;

      /* Driver registration */
      try  {
          Class.forName("oracle.jdbc.driver.OracleDriver");
      }
      catch(Exception e)  {
          System.err.println("Driver unavailable: "+e);
      }
```

Loading the driver

Example: selecting suppliers

```
try {
   /* Connection to the database */
  conn=DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe",
          “user123”,“pwd123");

  /* Creation of a statement for immediate commands */
  stmt = conn.createStatement();

  /* Assembling a query */
  VarCity =argv[0];
  query="SELECT SId, NEmployees FROM S WHERE City = '"+VarCity+"'";

  /* Execution of the query */
  rs=stmt.executeQuery(query);
```

Example: selecting suppliers

```
try {
    /* Connection to the database */
   conn=DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe",
          "user123","pwd123");

   /* Creation of a statement for immediate commands */
   stmt = conn.createStatement();

   /* Assembling a query */
   VarCity =argv[0];
   query="SELECT SId, NEmployees FROM S WHERE City = '"+VarCity+"'";

   /* Execution of the query */
   rs=stmt.executeQuery(query);
```

Creation of a statement

Example: selecting suppliers

```
try {
     /* Connection to the database */
    conn=DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe",
              "user123","pwd123");

    /* Creation of a statement for immediate commands */
    stmt = conn.createStatement();

    /* Assembling a query */
    VarCity =argv[0];
    query="SELECT SId, NEmployees FROM S WHERE City = '"+VarCity+"'";

    /* Execution of the query */
    rs=stmt.executeQuery(query);
```

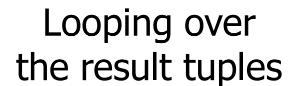
Example: selecting suppliers

```
try {
    /* Connection to the database */
   conn=DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe",
          "user123","pwd123");

   /* Creation of a statement for immediate commands */
   stmt = conn.createStatement();

   /* Assembling a query */
   VarCity =argv[0];
   query="SELECT SId, NEmployees FROM S WHERE City = '"+VarCity+"'";

   /* Execution of the query */
   rs=stmt.executeQuery(query);
```

Immediate query execution

Example: selecting suppliers

```
        System.out.println("Suppliers based in "+VarCity);
        /* Scan tuples in the result */
        while (rs.next()) {
          /* Print the current tuple */
           System.out.println(rs.getString("SId")+","+rs.getInt("NEmployees"));
        }
        /* Close resultset, statement and connection */
        rs.close();
        stmt.close();
        conn.close();
      }
      catch(Exception e) {
         System.err.println("Error: "+e);
      }
    }
  }
```



Example: selecting suppliers

```
        System.out.println("Suppliers based in "+VarCity);
        /* Scan tuples in the result */
        while (rs.next()) {
          /* Print the current tuple */
           System.out.println(rs.getString("SId")+","+rs.getInt("NEmployees"));
        }
        /* Close resultset, statement and connection */
        rs.close();
        stmt.close();
        conn.close();
      }
      catch(Exception e) {
          System.err.println("Error: "+e);
      }
    }
  }
```

Closing resultset, statement and connection

Updatable ResultSet

- It is possible to create an updatable ResultSet
  - the execution of updates on the database is more efficient
  - it is similar to an updatable cursor
    - there must be a one-to-one correspondence between the tuples in the result set and the tuples in the database tables

DBMG

Defining a transaction

- Connections are implicitly created with the *auto-commit mode* enabled
  - after each successful execution of an SQL command, a commit is automatically executed

Defining a transaction

- Connections are implicitly created with the *auto-commit mode* enabled
  - after each successful execution of an SQL command, a commit is automatically executed
- When it is necessary to execute a commit only after a sequence of SQL commands has been successfully executed
  - *a single* commit is executed after the execution of all commands
  - the commit must be managed in a *non automatico* fashion

Managing transactions

- The commit mode can be managed by invoking the setAutoCommit() method on the connection

```
void setAutoCommit(boolean autoCommit);
```

  - parameter autoCommit
    - true to enable autocommit (default)
    - false to disable autocommit

Managing transactions

- If autocommit is disabled
  - commit and rollback operations must be *explicitly* requested by the programmer
    - commit

      ```
      void commit();
      ```
    - rollback

      ```
      void rollback();
      ```
  - such methods are invoked on the corresponding connection

SQL for applications

Stored procedures

DBMG

Stored procedures

- A stored procedure is a function or a procedure defined inside the DBMS
  - it is stored in the data dictionary
    - it is part of the database schema
- It may be used like a predefined SQL command
  - it may have execution parameters
- It contains both application code and SQL commands
  - application code and SQL commands are tightly coupled to each other

Stored procedures: language

- The language used to define a stored procedure
  - is a procedural extension of the SQL language
  - depends on the DBMS
    - different products may offer different languages
    - the expressiveness of the language may vary according to the product

DBMG

Stored procedures: execution

- Stored procedures are integrated in the DBMS
  - server-side approach
- Performance is better compared to embedded SQL and CLI
  - each stored procedure is compiled and optimized *only once*
    - immediately after its definition
    - or when it is invoked for the first time

DBMG

Languages for stored procedures

- Different languages are available to define stored procedures
  - PL/SQL
    - Oracle
  - SQL/PL
    - DB2
  - Transact-SQL
    - Microsoft SQL Server
  - PL/pgSQL
    - PostgreSQL

Connection to the DBMS

- No connection to the DBMS is needed from within a stored procedure
  - the DBMS executing the SQL commands also stores and executes the stored procedure

DBMG

Managing SQL commands

- It is possible to reference variables or parameters in the SQL commands used in stored procedures
  - the syntax depends on the language used
- To read the result of a query that returns a set of tuples
  - a cursor must be defined
    - similar to embedded SQL

Stored procedures in Oracle

- Creazione di una stored procedure in Oracle

  CREATE [OR REPLACE] PROCEDURE *StoredProcedureName*
  [(*ParameterList*)]
  IS (*SQLCommand* | *PL/SQL code*);

- A stored procedure may be associated with
  - a single SQL command
  - a block of code written in PL/SQL

Parameters

- Each parameter in the *parametersList* list is specified in the form

parameterName [IN|OUT|IN OUT] [NOCOPY] *dataType*

- *parameterName*
  - name associated to the parameter
- *dataType*
  - type of the parameter
  - the SQL are used
- the keywords IN, OUT, IN OUT e NOCOPY specify that can be executed on the parameter
  - default IN

DBMG

Parameters

- Keyword IN
  - only-reading parameter
- Keyword OUT
  - only writing parameter
- Keyword IN OUT
  - the parameter can be both read and written in the stored procedure
- For the OUT and IN OUT parameters the final value is assigned only when the procedure ends correctly
  - the keyword NOCOPY allows to directly write the parameter during the execution of the stored procedure

Basic structure of a PL/SQL procedure

- Each PL/SQL block present in the body of a stored procedure must have the following structure

```
[variablesandCursorDeclaration]
BEGIN
codeToExecute
[EXCEPTION exceptionsManagementCode]
END;
```

PL/SQL Language

- The PL/SQL language is a procedural language
  - has some of the classical procedural languages commands
    - IF-THEN-ELSE control structures
    - cycles
  - has instruments for
    - the execution of SQL commands
    - scan results
      - cursors
- The SQL commands
  - are normal commands of the PL/SQL language
    - are not preceded by keywords
    - are not parameters of functions or procedures

Example: update command

- Update of the city of the supplier identified by the value present in the *supplierCode* parameter with the value present in *newCity*

```
CREATE PROCEDURE updateCity(supplierCode VARCHAR(5),
newCity VARCHAR(15))
IS
BEGIN
        UPDATE S SET City=newCity
        WHERE codS=supplierCode;
END;
```

Cursors in PL/SQL

- Declaration

  CURSOR *cursorName* IS *interrogazioneSQL* [FOR UPDATE];

- Opening

  OPEN *cursorName*;

- Reading of the next tuple

  FETCH *cursorName* INTO *VariablesList*;

- Closing

  CLOSE *cursorName*;

Example: selecting suppliers

- Show the code and the number of employees of the suppliers whose city is contained in the *VarCity* parameter

Example: selecting suppliers

```
CREATE PROCEDURE CitySuppliers(VarCity IN S.City%Type) IS
/*
      Definition of variables and cursors
*/
codeS    S.SId%Type;
NumEmployees   S.#Employees%Type;

CURSOR selectedSuppliers IS
SELECT SId,#Employees FROM S WHERE City = VarCity;

BEGIN
  DBMS_OUTPUT.PUT_LINE('Suppliers based in '||VarCity);
  /*
    Cursor opening
  */
  OPEN selectedSuppliers;
```

Example: selecting suppliers

```
/*
   Analysis of the data selected by the query
*/

LOOP
    FETCH selectedSuppliers INTO codeS, NumEmployees;
    /*
        Exit from the cycle when there are no more tuples to check
    */
    EXIT WHEN selectedSuppliers%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE(codeS||','||NumEmployees);
END LOOP;

/*
   Cursor closing
 */
 CLOSE selectedSuppliers;
END;
```

Example: selecting suppliers

```
CREATE PROCEDURE CitySuppliers(VarCity IN S.City%Type) IS
/*
      Definition of variables and cursors
*/
codeS    S.SId%Type;
NumEmployees   S.#Employees%Type;

CURSOR selectedSuppliers IS
SELECT SId,#Employees FROM S WHERE City = VarCity;

BEGIN
  DBMS_OUTPUT.PUT_LINE('Suppliers based in '||VarCity);
  /*
     Cursor opening
  */
  OPEN selectedSuppliers;
```

Example: selecting suppliers

```
CREATE PROCEDURE CitySuppliers(VarCity IN S.City%Type) IS
/*
      Definition of variables and cursors
*/
codeS    S.SId%Type;
NumEmployees   S.#Employees%Type;

CURSOR selectedSuppliers IS
SELECT SId,#Employees FROM S WHERE City = VarCity;

BEGIN
  DBMS_OUTPUT.PUT_LINE('Suppliers based in '||VarCity);
  /*
     Cursor opening
  */
  OPEN selectedSuppliers;
```

Example: selecting suppliers

```
CREATE PROCEDURE CitySuppliers(VarCity IN S.City%Type) IS
/*
      Definition of variables and cursors
*/
codeS    S.SId%Type;
NumEmployees   S.#Employees%Type;

CURSOR selectedSuppliers IS
SELECT SId,#Employees FROM S WHERE City = VarCity;

BEGIN
  DBMS_OUTPUT.PUT_LINE('Suppliers based in '||VarCity);
  /*
     Cursor opening
  */
  OPEN selectedSuppliers;
```

Example: selecting suppliers

```
CREATE PROCEDURE CitySuppliers(VarCity IN S.City%Type) IS
/*
       Definition of variables and cursors
*/
codeS    S.SId%Type;
NumEmployees   S.#Employees%Type;

CURSOR selectedSuppliers IS
SELECT SId,#Employees FROM S WHERE City = VarCity;

BEGIN
  DBMS_OUTPUT.PUT_LINE('Suppliers based in '||VarCity);
  /*
     Cursor opening
  */
  OPEN selectedSuppliers;
```

Example: selecting suppliers

```
/*
   Analysis of the data selected by the query
*/

LOOP
    FETCH selectedSuppliers INTO codeS, NumEmployees;
    /*
        Exit from the cycle when there are no more tuples to check
    */
    EXIT WHEN selectedSuppliers%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE(codeS||','||NumEmployees);
END LOOP;

/*
   Cursor closing
 */
 CLOSE selectedSuppliers;
END;
```

Example: selecting suppliers

```
/*
   Analysis of the data selected by the query
*/

LOOP
    FETCH selectedSuppliers INTO codeS, NumEmployees;
    /*
        Exit from the cycle when there are no more tuples to check
    */
    EXIT WHEN selectedSuppliers%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE(codeS||','||NumEmployees);
END LOOP;

/*
   Cursor closing
 */
 CLOSE selectedSuppliers;
END;
```

Cursor closing

SQL for applications

Comparison of alternatives

DBMG

Embedded SQL, CLI and Stored procedures

- The techniques proposed for the integration of the SQL language with applications have different features
- There is no winner: no one approach is always better than the others
  - it depends on the type of application
  - it depends on the characteristics of the databases
    - distributed, heterogeneous
- Mixed solutions may be adopted
  - invoking a stored procedure through CLI or embedded SQL

Embedded SQL vs. Call Level Interface

- Embedded SQL
  - (+) it precompiles static SQL queries
    - more efficient
  - (-) it depends on the adopted DBMS and operating system
    - due to the presence of a compiler
  - (-) it normally does not allow access to multiple databases at the same time
    - or it is a complex operation

Embedded SQL vs. Call Level Interface

- Call Level Interface
  - (+) independent of the adopted DBMS
    - only at compile time
      - the communication library (driver) implements a standard interface
      - the internal mechanism depends on the DBMS
    - the driver is loaded and invoked dynamically at runtime
  - (+) it does not require a precompiler

Embedded SQL vs. Call Level Interface

- Call Level Interface
  - (+) it allows access to multiple databases from within the same application
    - databases may be heterogeneous
  - (-) it uses dynamic SQL
    - lower efficiency
  - (-) it usually supports a subset of the SQL language

Stored procedures vs. client-side approaches

- Stored procedures
  - (+) greater efficiency
    - it exploits the tight integration with the DBMS
    - it reduces data exchange over the network
    - procedures are precompiled

Stored procedures vs. client-side approaches

- Stored procedures
  - (-) they depend on the DBMS
    - use of the DBMS ad-hoc language
    - usually not portable from one DBMS to another
  - (-) languages offer fewer functionalities than traditional languages
    - no functions available to create complex data visualizations of results
      - graphs and report
    - limited input management

Stored procedures vs. client-side approaches

- Client-side approaches
  - (+) based on traditional programming language
    - well known to programmers
    - more efficient compilers
    - wide range of input and output management functions
  - (+) greater independence from the adopted DBMS when writing code
    - only true of CLI-based approaches
  - (+) possibility to access heterogeneous databases

Stored procedures vs. client-side approaches

- Client-side approaches
  - (-) lower efficiency
    - lower degree of integration with the DBMS
    - compilation of SQL commands at runtime
      - especially for CLI-based approaches

DBMG