Database Programming Approaches

- Embedded commands:
  - Database commands are **embedded** in a general-purpose programming language
- Library of database functions:
  - Available to the host language for database calls; known as an **API** (Application Program Interface)
  - *e.g., JDBC, ODBC, PHP, Python*
- **A brand new, full-fledged language**
  - **e.g., Oracle PL/SQL, Postgres PL/pgSQL**
  - **Procedural Language extensions to SQL**

External Language Functions/Procedures

- Declaring external language procedures and functions
  - In C/C++


```
create procedure author_count_proc (in title varchar(20),
                                    out count integer)
language C
external name ' /usr/db/bin/author_count_proc'
```

  - In Java

```
create function author_count ( title varchar(20) )
   returns integer
language Java
external name '/usr/db/bin/author_count.jar'
```

Example in Oracle for C/C++

- CREATE LIBRARY [*schema_name*] <*library_name*> {IS | AS} <'*file_path*'> [AGENT <'*agent_link*'>];
  - `CREATE LIBRARY C_utils AS '/DLLs/utils.so';`
- Find the greatest common divisor of x and y x

```
CREATE FUNCTION gcd (
    x BINARY_INTEGER, y BINARY_INTEGER)
RETURN BINARY_INTEGER
AS EXTERNAL LIBRARY c_utils
NAME "c_gcd".              -- quotes preserve lower
case LANGUAGE C;
```


External Routines: Performance Vs. Security

- Benefits of external language functions/procedures:
  - more efficient for many operations, and
  - more expressive power
- Drawbacks
  - Code to implement function may need to be executed in the database system's address space
    - risk of accidental corruption of database structures
    - security risk, allowing users access to unauthorized data

  - Use *sandbox* techniques
    - that is use a safe language like Java
- Direct execution in the database system's space is used when efficiency is more important than security

Disadvantages of stored procedures

- Slowness in software development because stored procedure programming requires specialized skills that many developers do not possess.
- Difficult to manage versions and hard to debug.
- May not be portable to other database management systems e.g., MySQL or Microsoft SQL Server.

Database Programming Approaches

- Embedded commands:
  - Database commands are **embedded** in a general-purpose programming language
- Library of database functions:
  - Available to the host language for database calls; known as an **API** (Application Program Interface)
  - *e.g., JDBC, ODBC, PHP, Python*
- A brand new, full-fledged language
  - e.g., Oracle PL/SQL, Postgres PL/pgSQL
  - **P**rocedural **L**anguage extensions to SQL

Embedded SQL (ESQL)

- SQL statements are embedded by enclosing them:
  - between "&SQL(" and ")";
  - between "EXEC SQL" and "END-EXEC"; or
  - between "EXEC SQL" and ";"
- E.g., **EXEC SQL** DELETE FROM STUDENT
  WHERE Name LIKE 'John %';
- Two types of statement-level embedding:
  - *Static SQL*: complete SQL statements
  - *Dynamic SQL*: statements are created during execution time

Structure of ESQL Programs

1. Define host data structures
2. Open database using EXEC SQL CONNECT dbname/username IDENTIFIED BY password
3. Start a transaction using EXEC SQL SET TRANSACTION command
4. Retrieve data using EXEC SQL SELECT and load into data structure that overlays row
5. Write data back to the database using EXEC SQL UPDATE, or INSERT, or DELETE
6. Terminate the transaction using either EXEC SQL COMMIT or EXEC SQL ROLLBACK
7. Close database using EXEC SQL DISCONNECT

ESQL

- The arguments used in an SQL statement could be *constants* or program *variables*
- Program variables within an SQL command are prefixed with "**:**" and declared within a DECLARE SECTION
- E.g.,

```
EXEC SQL BEGIN DECLARE SECTION
char student_name[20];
EXEC SQL END DECLARE SECTION

cout << "Please Enter Student Name to be deleted" << endl;
cin >> (char *) student_name;
EXEC SQL DELETE FROM STUDENT
        WHERE Name = : student_name;
```


Host Data Structure

- Structures/records (e.g., C struct) must match tuple formats *exactly*
  - field order is important
  - strings in C/C++ are terminated with NULL character
- SQLCODE in SQL1 is an integer variable containing the Status Code returned by SQL/DBS
  - Zero (0) if SQL command is successful
  - Nonzero positive if SQL generates a warning
    - 100: data not found
  - Negative if SQL command fails (error)
    - -913: resource deadlock
- SQLSTATE in SQL2 is a 5-character string
    - 02000: data not found

NULL Values & Indicators

- INDICATORS are used to hold the status of variables and test for NULL values.
- An indicator is associated with each variable:
  - Short integer (2 bytes).
    - -1 : indicates NULL value
    - 0 : indicates valid data value.
  - Always *check* indicator when reading.
  - Always *set* indicator when writing.
- Note that each field in a struct is a separate variable. That is, a 4 field struct is associated with 4 indicators
- Indicators could be a struct or an array depending on the implementation

Host Data for Single Retrieval

```
Struct {
   int sid;
   VARCHAR   student_name[UNAME_LEN];
   char     major[5];
} student_rec;

Struct {
   short     sid_ind;
   short     student_name_ind;
   short     major_ind;
} student_rec_ind;
```

ESQL Cursors

- If more than one tuples can be selected, then tuples must be processed one at a time by means of a cursor
  - This is similar to the record-at-a-time processing
- A cursor is a "pointer" to a tuple in a result of a query
  - Current tuple w.r.t. a cursor is the tuple pointed by the cursor
- DECLARE <cursor-name> CURSOR FOR <query>
  - It defines a query and associates a cursor with it
- OPEN <cursor-name> brings the query result from the DB and positions the cursor before the first tuple
- CLOSE cursor-name closes the named cursor and deletes the associated result table

Cursor Retrieval

- **Sequential Access:**
  FETCH <cursor-name> INTO <variable-list>;
  - copies into variables the current tuple and advances the cursor
  - Use SQLCODE, SQLSTATE, or WHENEVER NOT FOUND to detect end of result table
- **Random Access:** (positioning of cursor)
  FETCH **orientation**
  FROM <cursor-name> INTO <variable-list>;
  - where **orientation**: NEXT (default), PRIOR, LAST, ABSOLUTE <offset>, RELATIVE <offset>

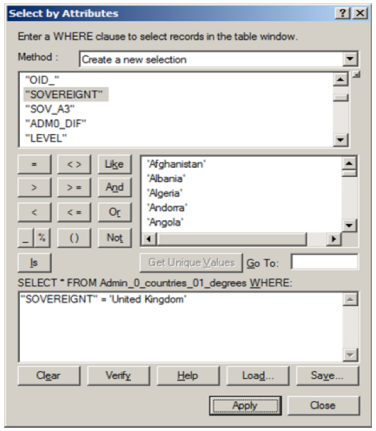
Cursor Retrieval Example

```
EXEC SQL DECLARE st_cursor CURSOR FOR
    SELECT SID, Name, Major
    FROM STUDENT
    WHERE Major = 'CS' ;
EXEC SQL OPEN st_cursor;
While (1) {
   EXEC SQL WHENEVER NOT FOUND DO break;
   EXEC SQL FETCH st_cursor
       INTO :student_rec INDICATOR :student_rec_ind;
  display_student(student_rec, student_rec_ind);
 };
EXEC SQL CLOSE st_cursor;
```


Dynamic SQL Statements

- An SQL statement is passed to DBMS in the form of a string to be interpreted and executed
- EXECUTE IMMEDIATE
- PREPARE, EXECUTE, USING
  - create/drop table
  - insert, delete, update
- Dynamic DECLARE CURSOR, DESCRIBE, OPEN, FETCH
  - select statement
- RELEASE

SQL Builder



Dynamic SQL: Delete Example

```
EXEC SQL BEGIN DECLARE SECTION
char student_name[20];
char sqltxt[100];
EXEC SQL END DECLARE SECTION

cout << "Please Enter Student Name to be deleted" << endl;
cin >> (char *) student_name;
strcpy("DELETE FROM STUDENT WHERE Name = '", sqltxt);
strcat(sqltxt, student_name);
strcat(sqltxt,"'");

EXEC SQL EXECUTE IMMEDIATE : sqltxt;
```


Dynamic SQL: Prepare-Execute-Using

- It compiles an SQL statement with parameters indicated with "?"

```
char sqltxt[] = "DELETE FROM STUDENT WHERE id = ?";
EXEC SQL PREPARE delcmd FROM : sqltxt;
```

- Using statement allows the passing of parameters

```
cout << "Please Enter Student Name to be deleted" << endl;
cin >> (char *) student_name;
EXEC SQL EXECUTE delcmd USING : student_name;
```

- Release statement

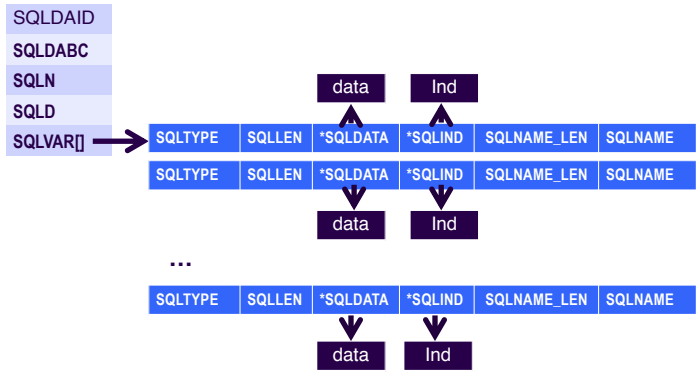
```
EXEC SQL RELEASE delcmd;
```


SQLDA

- Need to declare a Description Area (SQLDA) to be used for communication with the DBMS

```
struct SQLDA_STRUCT {
    char SQLDAID[8];
    int SQLDABC;
    short SQLN;        /* Max# of variables in SQLVAR */
    short SQLD;        /* # of select list items   */
    struct {
        short SQLTYPE;      /* SQL Data type.       */
        short SQLLEN;       /* SQL Data length.     */
        char *SQLDATA;      /* ptr: SQL data.       */
        short *SQLIND;      /* ptr: SQL indicator var. */
        short SQLNAME_LEN;  /* length of SQL name.  */
        char SQLNAME[30];   /* SQL name.            */
    } SQLVAR[];        /* Variable length SQLVARARY.  */
};
```

SQLDA



SQLJ: SQL-Java

- Semi-static version of embedded SQL in Java
- SQL statements are introduced with: *#sql*

```
#sql { delete from STUDENT where SID = :stid };
#sql { insert into STUDENT (SID) values (165) };
```

- *Iterator* object supports the notion of cursor

```
#sql iterator ST_Cursor (Integer Sid, String Name);
ST_Cursor stCursor;
#sql stCursor = { SELECT SID, Name  INTO :Sid, :Name
                  FROM STUDENT WHERE Major = 'none' };
while (stCursor.next()) {
  System.out.println(stCursor.Sid() + ", " + stCursor.Name()); }
stCursor.close();
```


SQLJ: Simple yet complete example

```
1.  import java.sql.*;      // you need this import for SQLException and other classes from JDBC
2.  import oracle.sqlj.runtime.Oracle;
3.  public class SingleRowQuery extends Base {
4.    public static void main(String[] args) {
5.      try {  connect();
6.             singleRowQuery(1);
7.      } catch (SQLException e) { e.printStackTrace(); }
8.    }
9.    public static void singleRowQuery(int id) throws SQLException {
10.     String fullname = null;
11.     String street = nul
12.     #sql {
13.       SELECT fullname,
14.         street INTO : OUT fullname,        //OUT is actually the default for INTO host variables
15.         : OUT street FROM customer WHERE ID = :id};
16.     System.out.println("Customer with ID = " + id);
17.     System.out.println();   System.out.println(fullname + " " + street);
18.   }
19. }
```