Database Applications

- SQL isn’t sufficient to write general applications.⇒ connect it with a general-purpose PL!
- Language considerations:
  ⇒ Library calls (CLI/ODBC)
  ⇒ Embedded SQL
  ⇒ Advanced persistent PL (usually OO)
- Client-server:
  ⇒ SQL runs on the server
  ⇒ Application runs on the client

Embedded SQL

- SQL Statements are **embedded** into a **host language** (C, C++, FORTRAN, ...)
- The application is **preprocessed** pure host language program + library calls
  - Advantages:
    * Preprocessing of (static) parts of queries
    * MUCH easier to use
  - Disadvantages:
    * Needs precompiler
    * Needs to be **bound** to a database

Development Process for Embedded SQL Applications
Embedded SQL (cont.)

Considerations:
- How much can SQL be parameterized?
  - How to pass parameters into SQL?
  - How to get results?
  - Errors?
- Static vs. dynamic SQL statements.

How much does the DBMS know about an application?
- precompiling: PREP
- binding: BIND

Application Structure

```c
main(int argc, char **argv)
{
    Declarations
    Connect to Database
    Do your work
    Process errors
    Commit/Abort and Disconnect
};
```

Declarations

- Include SQL communication area:
  ```sql
  EXEC SQL INCLUDE SQLCA;
  ```
  it defines:
  - the return code of SQL statements (sqlcode)
  - the error messages (if any)
  - ... you can’t live without it.
- SQL statements inserted using magic words
  ```sql
  EXEC SQL <sql statement> ;
  ```

Host Variables

- are used to pass values between a SQL and the rest of the program:
  - parameters in SQL statements:
    - communicate **single values** between
      SQL a statement and host language variables
  - must be declared within SQL declare section:
    ```sql
    EXEC SQL BEGIN DECLARE SECTION;
    declarations of variables to be used
    in SQL statements go here
    EXEC SQL END DECLARE SECTION;
    ```
  - can be used in the EXEC SQL statements:
    - to distinguish them from SQL identifiers
      they are preceded by `:` (colon)
Errors

What if a SQL statement fails?

- check sqlcode != 0
- use “exception” handling:

```
EXEC SQL WHENEVER SQLERROR GO TO lbl;
EXEC SQL WHENEVER SQLWARNING GO TO lbl;
EXEC SQL WHENEVER NOT FOUND GO TO lbl;
⇒ designed for COBOL (lbl has to be in scope).
```

Dummy Application (DB2)

```
#include <stdio.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {
  EXEC SQL BEGIN DECLARE SECTION;
      char db[6] = "DBCLASS";
  EXEC SQL END DECLARE SECTION;
  printf("Sample C program: CONNECT\n");
  EXEC SQL WHENEVER SQLERROR GO TO error;
  EXEC SQL CONNECT TO :db;
  printf("Connected to DB2\n");
  // do your stuff here
  EXEC SQL COMMIT;
  EXEC SQL CONNECT reset;
  exit(0);
error:
  check_error("My error", &sqlca);
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL ROLLBACK;
  EXEC SQL CONNECT reset;
  exit(1);
}
```

Preparing your Application (DB2)

1. write the application in a file called `<name>.sqc`
2. preprocess the application:
```
db2 prep <name>.sqc
```
3. compile the application:
```
cc -c -O <name>.c
```
4. link with DB2 libraries:
```
cc -o <name> <name.o> -L... -l...
```
5. run it:
```
./<name> [arguments]
```

Typically comes with a Makefile
⇒ sets options
⇒ knows the path(s) and libraries

Dummy Application (Oracle)

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[]) {
  EXEC SQL BEGIN DECLARE SECTION;
      char user[6] = "DBCLASS";
      char pwd[10];
  EXEC SQL END DECLARE SECTION;
  printf("Sample C program: CONNECT\n");
  strncpy(pwd, getpass("Password: "), 10);
  EXEC SQL WHENEVER SQLERROR GO TO error;
  EXEC SQL CONNECT :user IDENTIFIED BY :pwd;
  printf("Connected to Oracle\n");
  // do your stuff here
  EXEC SQL COMMIT RELEASE;
  exit(0);
error:
  sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '\0';
  printf("MyError %s\n", sqlca.sqlerrm.sqlerrmc);
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL ROLLBACK RELEASE;
  exit(1);
}
```
Example of a build (DB2)

bash$ make NAME=sample1
db2 connect to DBCLASS

Database server = DB2/SUN 6.1.0
SQL authorization ID = DAVID
Local database alias = DBCLASS

db2 prep sample1.sqc bindfile
LINE MESSAGES FOR sample1.sqc

--- ---------------------------------------------------
SQL0060W The "C" precompiler is in progress.
SQL0091W Precompilation or binding was ended with "0" errors and "0" warnings.
db2 bind sample1.bnd
LINE MESSAGES FOR sample1.bnd

--- ---------------------------------------------------
SQL0061W The binder is in progress.
SQL0091N Binding was ended with "0" errors and "0" warnings.
db2 connect reset
DB20000I The SQL command completed successfully.
cc -I/usr/db2/include -c sample1.c
cc -I/usr/db2/include -o sample1 sample1.o util.o
-L/usr/db2/lib -R/usr/db2/lib -ldb2

"Real" SQL Statements

So far we introduced only the surrounding infrastructure. Now for the real SQL statements:

- simple statements:
  - "constant" statements
  - statements with parameters
  - statements returning a single tuple

- general queries with many answers

- dynamic queries (not covered here)

Simple Application

Write a program that for each publication id supplied as an argument prints out the title of the publication:

```c
main(int argc, char *argv[]) { ... printf("Connected to DB2\n"); for (i=1; i<argc; i++) { strncpy(pubid,argv[i],8);
EXEC SQL WHENEVER NOT FOUND GO TO nope;
EXEC SQL SELECT title INTO :title FROM publication WHERE pubid = :pubid;
printf("%10s: %s\n",pubid,title);
continue;
nope:
printf("%10s: *** not found *** \n",pubid);
}; ...
}...
```
Simple Application (cont.)

bash$ ./sample2 ChTo98 nopubid
Sample C program: SAMPLE2
Connected to DB2
ChTo98: Temporal Logic in Information Systems
nopubid: *** not found ***

⇒ it is important that at most one title is returned for each pubid.

NULLs and Indicator Variables

- what if a host variable is assigned a NULL?
  ⇒ not a valid value in the datatype
  ⇒ ESQL uses an extra Indicator variable, e.g.:

```sql
smallint ind;
SELECT firstname INTO :firstname
INDICATOR :ind
FROM ...
```
then if ind < 0 then firstname is NULL
- if the indicator variable is not provided and the result is a null we get a run-time error
- the same rules apply for host variables in updates.

Impedance Mismatch

What if we EXEC SQL a query and it returns more than one tuple?

1. Declare the cursor:
   ```sql
   EXEC SQL DECLARE <name> CURSOR
   FOR <query>;
   ```
2. Iterate over it:
   ```sql
   EXEC SQL OPEN <name>;
   EXEC SQL WHENEVER NOT FOUND GO TO end;
   for (;;) {
     <set up host parameters>
     EXEC SQL FETCH <name>
     INTO <host variables>;
     <process the fetched tuple>
   }
   end:
   EXEC SQL CLOSE <name>;
   ```

Application with a Cursor

Write a program that lists all author names and publication titles with author name matching a pattern given as an argument:

```c
main(int argc, char *argv[]) {
  ...
  strncpy(apat,argv[1],8);
  EXEC SQL DECLARE author CURSOR
  FOR SELECT name, title
  FROM author , wrote, publication
  WHERE name LIKE :apat
  AND aid=author AND pubid=publication;
  EXEC SQL OPEN author;
  EXEC SQL WHENEVER NOT FOUND GO TO end;
  for (;;) {
    EXEC SQL FETCH author INTO :name, title;
    printf("%10s -> %20s: %s\n",apat,name,title);
  }
  end:
  ...
}
```
Application with a Cursor (cont.)

bash$ ./sample3 "%"
Sample C program: SAMPLE3
Connected to DB2
  % -> Toman, David : Temporal Logic in Information
  % -> Toman, David : Datalog with Integer Periodic
  % -> Toman, David : Point-Based Temporal Extensio
  % -> Chomicki, Jan : Logics for Databases and Info
  % -> Chomicki, Jan : Datalog with Integer Periodic
  % -> Chomicki, Jan : Temporal Logic in Information
  % -> Saake, Gunter : Logics for Databases and Info
bash$ ./sample3 "T%"
Sample C program: SAMPLE3
Connected to DB2
  T% -> Toman, David : Temporal Logic in Information
  T% -> Toman, David : Datalog with Integer Periodic
  T% -> Toman, David : Point-Based Temporal Extensio

Call Level Interface/ODBC

An interface built on a library calls:
- Applications are developed without access to the DB (and without additional tools: no precompilation)
- incorporates ODBC (MS) and X/Open standards
- but it is harder to use and doesn’t allow preprocessing (e.g., no checking of your SQL code and data types)

Three fundamental objects in an ODBC program:
- Environments
- Connections
- Statements

Connect and Disconnect

```c
int main()
{
    SQLHENV henv;
    SQLHDBC hdbc;
    SQLRETURN rc;
    SQLCHAR server[SQL_MAX_DSN_LENGTH + 1] = "DBCLASS";
    SQLCHAR uid[19] = "<your uid>";
    SQLCHAR pwd[31] = "<your password>";

    SQLAllocEnv(&henv);
    SQLAllocConnect(henv, &hdbc);
    rc = SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
    if (rc != SQL_SUCCESS) {
        printf("Error connecting to %s\n", server); exit(1);
    } else printf("Connected to %s\n", server);

    /* DO SOMETHING HERE */

    SQLDisconnect(hdbc);
    SQLFreeConnect(hdbc);
    SQLFreeEnv(henv);
}
```

Summary

- **Declarations:**
  ```sql
  EXEC SQL INCLUDE SQLCA;
  EXEC SQL BEGIN DECLARE SECTION;
  <host variables here>
  EXEC SQL END DECLARE SECTION;
  ```

- **Simple statements:**
  ```sql
  EXEC SQL <SQL statement>;
  ```

- **Queries (with multiple answers)**
  ```sql
  EXEC SQL DECLARE <id> CURSOR FOR <qry>;
  EXEC SQL OPEN <id>;
  do {
    EXEC SQL FETCH <id> INTO <vars>;
  } while (SQLCODE == 0);
  EXEC SQL CLOSE <id>;
  ```

- Don’t forget to check errors!!
Errors

- SQLxxx functions return error codes
  ⇒ similar to libc functions
  ⇒ we should check them after every SQLxxx call
- the actual return codes:
  - SQL_SUCCESS
  - SQL_ERROR
- use the SQLError function to get sensible messages

SQL Statements

...and what we can do with them:

- SQLAllocStmt (allocates object)
- SQLExecDirect (execute)
- SQLPrepare (compile statement)
- SQLEXecute (execute compiled statement)
- SQLSetParam (initialize a procedure parameter)
- SQLNumResultCols (number of result columns)
- SQLBindCol ("host variables" in ODBC)
- SQLGetData (obtaining values of result columns)
- SQLFetch (cursor access in ODBC)
- SQLError (obtains diagnostics)
- SQLRowCount (number of affected rows)
  ...
- SQLFreeStmt (frees object)

Parameters

1. parameter markers
   - ‘?’ in the text of the query
   - SQLNumParams
   - SQLBindParameter

2. results of queries
   ⇒ specified by the number of resulting columns
   - SQLNumResultsCol
   - SQLDescribeCol
   - SQLBindCol or SQLGetData

3. number of affected tuples (updates):
   - SQLRowCount

Example

```sql
SQLCHAR stmt[] = "UPDATE author SET url = ? WHERE aid = ?";
SQLINTEGER aid;
SQLCHAR s[70];
SQLINTEGER ind;
rc = SQLAllocStmt(hdbc, &hstmt);
rc = SQLPrepare(hstmt, stmt, SQL_NTS);
printf("Enter Author ID: "); scanf("%ld", &aid);
printf("Enter Author URL: "); scanf("%s", s);
rc = SQLBindParameter(hstmt, 1,
                      SQL_PARAM_INPUT, SQL_C_CHAR,
                      SQL_CHAR, 0, 0, s, 70, &ind);
rc = SQLBindParameter(hstmt, 2,
                      SQL_PARAM_INPUT, SQL_C_SLONG,
                      SQL_INTEGER, 0, 0, &aid, 0, NULL);
rc = SQLExecute(hstmt);
```
Answers

How to get output values from a statement

- number of affected: SQLRowCount
- answers to queries:
  1. bind variables before execution: SQLBindCol
  2. get values after execution: SQLGetData
- get next tuple: SQLFetch
  the result of SQLFetch is just a result code!

A Query with SQLBindCol

```c
SQLCHAR sqlstmt[] = "SELECT pubid, title FROM publication";
SQLINTEGER rows;
struct { SQLINTEGER ind;
    SQLCHAR s[70];
} pubid, title;
rc = SQLAllocStmt(hdbc, &hstmt);
rc = SQLExecDirect(hstmt, sqlstmt, SQL_NTS);
rc = SQLBindCol(hstmt, 1, SQL_C_CHAR,
                (SQLPOINTER)pubid.s, 8, &pubid.ind);
rc = SQLBindCol(hstmt, 2, SQL_C_CHAR,
                (SQLPOINTER)title.s, 70, &title.ind);
while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
    printf("%-8.8s %-70.70s\n", pubid.s, title.s);
rc = SQLRowCount(hstmt, &rows);
printf(" %d rows selected\n", rows);
rc = SQLFreeStmt(hstmt, SQL_DROP);
```

Transactions

- transaction start:
  ⇒ implicitly using one of
    SQLPrepare,
    SQLExecute,
    SQLExecDirect, etc.
- transaction end:
  SQLTransact(henv, hdbc, what)
  where
    what = SQL_COMMIT, or
    what = SQL_ROLLBACK

Summary

- CLI/ODBC can do everything Embedded SQL can.
- However, all statements are dynamic
  ⇒ no precompilation
  ⇒ explicit binding of parameters (user has to make types match!)
- An almost standard (ODBC, X/Open)
  ⇒ independence on DBMS
  ⇒ but: the standard has 100’s of functions
A stored procedure executes application logic directly inside the DBMS process.

Possible implementations:
- invoke externally-compiled application
- SQL/PSM (or vendor-specific language)

Possible advantages of stored procedures:
1. minimize data transfer costs
2. centralize application code
3. logical independence

A Stored Procedure Example: Atomic-Valued Function

```
CREATE FUNCTION sumSalaries(dept CHAR(3))
RETURNS DECIMAL(9,2)
LANGUAGE SQL
RETURN
    SELECT sum(salary)
    FROM employee
    WHERE workdept = dept
```

A Stored Procedure Example: Table-Valued Function

```
CREATE FUNCTION deptSalariesF(dept CHAR(3))
RETURNS TABLE(salary DECIMAL(9,2))
LANGUAGE SQL
RETURN
    SELECT salary
    FROM employee
    WHERE workdept = dept
```

```
db2 => SELECT deptno, sumSalaries(deptno) AS sal \
    => FROM department
```

```
DEPTNO   SAL
-------- --------
A00      128500.00
B01      41250.00
C01      90470.00
D01      -
D11      222100.00
D21      150920.00
E01      40175.00
E11      104990.00
E21      95310.00
```

9 record(s) selected.
A Stored Procedure Example: Table-Valued Function

db2 => SELECT * FROM TABLE \
    => (deptSalariesF(CAST('A00' AS CHAR(3)))) AS s

SALARY
--------
52750.00
46500.00
29250.00

3 record(s) selected.

A Stored Procedure Example: Branching

CREATE PROCEDURE UPDATE_SALARY_IF
    (IN employee_number CHAR(6), INOUT rating SMALLINT)
    LANGUAGE SQL
BEGIN
    DECLARE not_found CONDITION FOR SQLSTATE '02000';
    DECLARE EXIT HANDLER FOR not_found
        SET rating = -1;
    IF rating = 1 THEN
        UPDATE employee
        SET salary = salary * 1.10, bonus = 1000
        WHERE empno = employee_number;
    ELSEIF rating = 2 THEN
        UPDATE employee
        SET salary = salary * 1.05, bonus = 500
        WHERE empno = employee_number;
    ELSE
        UPDATE employee
        SET salary = salary * 1.03, bonus = 0
        WHERE empno = employee_number;
    END IF;
END