Application Programming and SQL

Spring 2018

School of Computer Science
University of Waterloo

Databases CS348

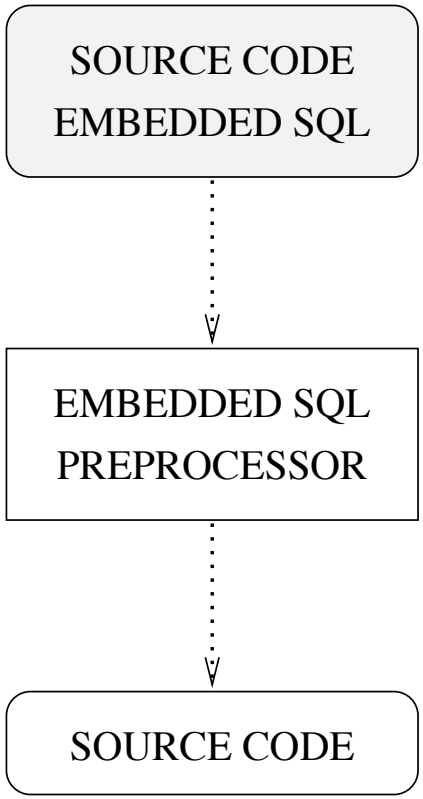
Database Applications

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

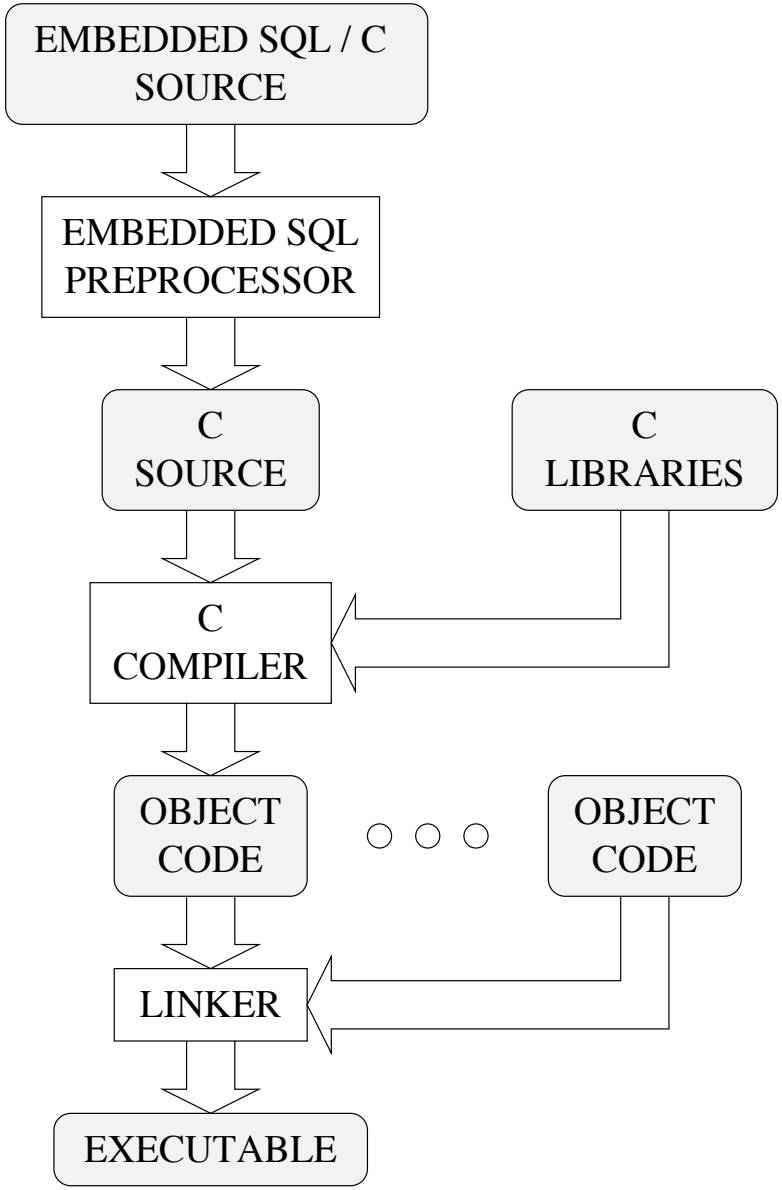
Embedded SQL

- 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



General structure



Embedded SQL (cont.)

- Considerations:

  $\Rightarrow$ How much can SQL be parameterized?

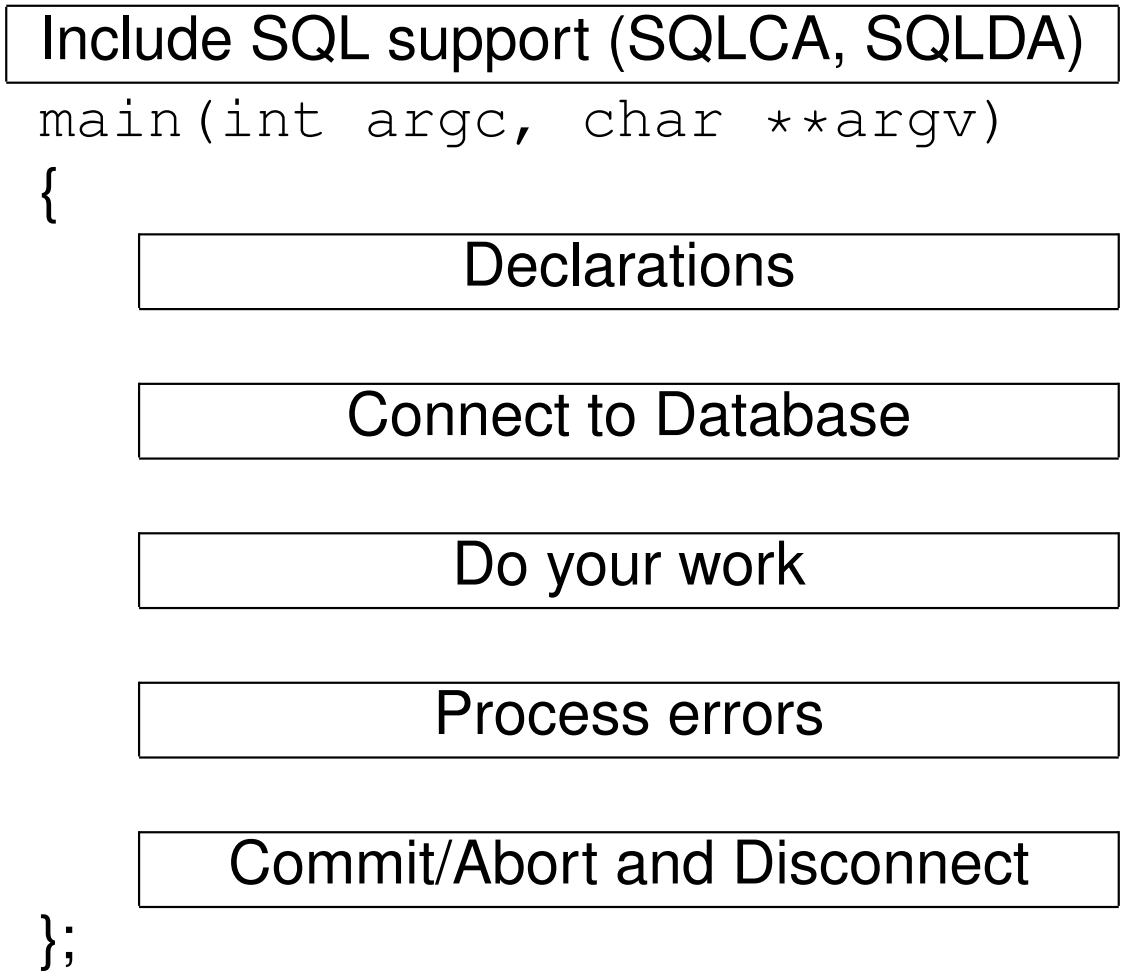
  - $*$ How to pass parameters into SQL?
  - $*$ How to get results?
  - $*$ Errors?

  $\Rightarrow$ Static vs. dynamic SQL statements.

- How much does the DBMS know about an application?

  $\Rightarrow$ precompiling: `PREP`
  $\Rightarrow$ binding: `BIND`

Application Structure



Declarations

- Include SQL communication area:

```
EXEC SQL  INCLUDE SQLCA;
```

  it defines:

  $\Rightarrow$ the return code of SQL statements (sqlcode)
  $\Rightarrow$ the error messages (if any)
  $\Rightarrow$ . . . you can't live without it.

- SQL statements inserted using magic words

```
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:

```
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:

  $\Rightarrow$ 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;
```

$\Rightarrow$ 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);
}
```

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);
}
```

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

$\Rightarrow$ sets options

$\Rightarrow$ knows the path(s) and libraries

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
```

Example

```
bash$ ./sample1
Sample C program: CONNECT
Connected to DB2
bash$
```

```
bash$ ./sample1
Sample C program: CONNECT
DB2 database error 0x80004005:  SQL30081N
A communication error has been detected.
Communication protocol being used: "TCP/IP".
...
SQLSTATE=08001
bash$
```

“Real” SQL Statements

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

- simple statements:

  $\Rightarrow$ “constant” statements
  $\Rightarrow$ statements with parameters
  $\Rightarrow$ 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:

```
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 ***
```

$\Rightarrow$ 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?

  $\Rightarrow$ not a valid value in the datatype
  $\Rightarrow$ ESQL uses an extra *Indicator* variable, e.g.:

```
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 an 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*:

```
EXEC SQL DECLARE <name> CURSOR
         FOR <query>;
```

2. Iterate over it:

```
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:

```
    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
```

Cursors and Updates

- cursors iterate over tuples in the answer . . . so you can *change* the tuple the cursor points to

  $\Rightarrow$ remember updating views? (same rules here)

- the value to be changed has to be specified in the declaration:

```
EXEC SQL DECLARE <name> CURSOR
         FOR <query>
         FOR UPDATE [ OF <attribs> ];
```

- the actual change:

```
EXEC SQL FETCH <cursor> INTO <vars>;
if <cond on variables>
  EXEC SQL UPDATE <cursor> SET ...
           WHERE CURRENT OF <name>;
```

  $\Rightarrow$ the `UPDATE` must match the cursor declaration.

Example

```
main(int argc, char *argv[]) {
   ...
   EXEC SQL DECLARE author CURSOR
         FOR SELECT name FROM author WHERE url IS NULL
             FOR UPDATE OF url;

   EXEC SQL OPEN author;
   EXEC SQL WHENEVER NOT FOUND GO TO end;
   for (;;) {
     EXEC SQL FETCH author INTO :name;
     printf("Author '%s' has no URL\n", name);
     printf("Enter new URL to fix or <cr> to delete: "); gets(url);
     if (strcmp(url,"")==0) {
       printf("Deleting '%s'\n",name);
       EXEC SQL DELETE FROM author
                WHERE CURRENT OF author;
     } else {
       printf("Seting URL for '%s' to '%s'\n",name,url);
       EXEC SQL UPDATE author
                SET url = :url
                WHERE CURRENT OF author;
     }
   };
end:
```

Summary

- Declarations:

```
EXEC SQL INCLUDE SQLCA;
EXEC SQL BEGIN DECLARE SECTION;
   <host variables here>
EXEC SQL END DECLARE SECTION;
```

- Simple statements:

```
EXEC SQL <SQL statement>;
```

- Queries (with multiple answers)

```
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!!

Stored Procedures

Idea

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: Atomic-Valued Function

```
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

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

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
```