Transaction Processing: Examples (Elmasri & Navathe 19.3)

- My wife and I share checking and savings bank accounts.
- Transactions:
  1. I transfer $500 from our savings account to our checking account.
  2. My wife and I visit different ATMs at exactly the same time to withdraw cash from our savings account.

Execution of Transaction 1(Simplified View):
I transfer $500 from our savings account to our checking account.
- **Read** savings balance.
- Subtract $500 from balance.
  - Assume balance is sufficient.
- **Read** checking balance.
- Add $500 to checking balance.
- **Write** new savings balance.
- **Write** new checking balance.

Execution of Transaction 1(Simplified View):
I transfer $500 from our savings account to our checking account. **Crash occurs.**
- **Read** savings balance.
- Subtract $500 from balance.
  - Assume balance is sufficient.
- **Read** checking balance.
- Add $500 to checking balance.
- **Write** new savings balance.
- **Write** new checking balance.

Execution of Transaction 2(Simplified view):
My wife and I withdraw cash at exactly the same time to from our savings account.
- **My transaction**
  - **Read** savings balance.
  - Subtract $100.
  - **Write** new savings balance.
  - Pay $100.
- **My wife’s transaction**
  - **Read** savings balance.
  - Subtract $200.
  - **Write** new savings balance.
  - Pay $200.

Execution of Transaction 2(Simplified view):
My wife withdraw cash at exactly the same time to from our savings account.
- **Read** savings balance (me).
- Subtract $100 (me).
- **Write** new savings balance (me).
- Pay $100 (me).
- **Read** savings balance (wife).
- Subtract $200 (wife).
- **Write** new savings balance (wife).
- Pay $200 (wife).

Execution of Transaction 2(Simplified view):
My wife withdraw cash at exactly the same time to from our savings account. **But what if the steps occur in this order?**
Transaction ACID Properties

- Atomicity
- Consistency preservation
- Isolation
- Durability or permanence

ACID: Atomicity

- Transaction is an atomic unit of processing;
- Performed in its entirety or not at all.
  - Transaction 1 (both or neither):
    - $100 removed from Savings
    - $100 added to checking

ACID: Consistency

- Transactions preserve consistency
  - Start in consistent state.
  - End in consistent state.

  It is assumed the code for processing the transaction and the database system enforcing integrity constraints maintain consistency of the database.

ACID: Isolation

- Transactions appear as though they were executed in isolation.
- That is one transaction must not interfere with another.
  - I withdraw $100.
  - My wife withdraws $200.
  - Neither we nor the bank loses or gains $100 or $200.

  Responsibility of the Concurrency Control system.

ACID: Durability

- Changes to database made by committed transaction must persist in spite of any failure.

  Responsibility of the Recovery System.

Transaction Processing

- Transaction
  - A sequence of database statements that must be executed atomically -- either all statements completed successfully or none of them

- Two types of transactions
  - Read Only: Only query statements
  - Read Write: Both query and update statements
**Transaction Commands**

**Set Transaction Type**
- EXEC SQL SET TRANSACTION READ ONLY;
- EXEC SQL SET TRANSACTION READ WRITE ONLY;
- Default is READ/WRITE.

**End Transaction**
- EXEC SQL COMMIT WORK RELEASE;
  - Commits changes and release locks
- EXEC SQL ROLLBACK WORK RELEASE;
  - Undoes changes and releases locks

**Example 4 - Read Only Transaction**

In the following example, as a store manager, you check sales activity for the day, the past week, and the past month by using a read-only transaction to generate a summary report. The report is unaffected by other users updating the database during the transaction.

```sql
EXEC SQL SET TRANSACTION READ ONLY;
EXEC SQL SELECT sum(saleamt) INTO :daily FROM sales
WHERE saledate = SYSDATE;
EXEC SQL SELECT sum(saleamt) INTO :weekly FROM sales
WHERE saledate > SYSDATE - 7;
EXEC SQL SELECT sum(saleamt) INTO :monthly FROM sales
WHERE saledate > SYSDATE - 30;
EXEC SQL COMMIT;
/* simply ends the transaction since there are no changes to make permanent */
```

**Example 5 - Read/Write Transaction**

```sql
/* Transfer $100 from checking to saving*/
DECLARE trans_flag = 0;
EXEC SQL SET TRANSACTION READ WRITE ONLY;
IF (sqlca.sqlcode = 0) trans_flag = 1;
EXEC SQL UPDATE checking SET balance = balance -100
WHERE acctno = :acctno;
EXEC SQL UPDATE saving SET balance = balance +100
WHERE acctno = :acctno;
IF (trans_flag = 1) EXEC SQL ROLLBACK
ELSE EXEC SQL COMMIT;
```

**Example 6 - Sqrerror**

If a WHENEVER SQLERROR GOTO statement branches to an error handling routine that includes a ROLLBACK statement, your program might enter an infinite loop if the ROLLBACK fails with an error. You can avoid this by coding WHENEVER SQLERROR CONTINUE before the ROLLBACK statement, as shown in the following example:

```sql
EXEC SQL WHENEVER SQLERROR GOTO sqerror;
FOR i :=...
    print("Employee number ");
    get(empnum);
    employee = atoi(empnum);
    print("Employee name ");
    get(empname);
EXEC SQL INSERT INTO emp (empnum, empname)
VALUES (empnum, empname);
```

**COMMIT Statement**

Use the COMMIT statement to make changes to the database permanent. Until changes are COMMITed, other users cannot access the changed data; they see it as it was before your transaction began. Specifically, the COMMIT statement:
- makes permanent all changes made to the database during the current transaction
- makes these changes visible to other users
- releases all locks
- ends the transaction

In the following example, you commit your transaction and disconnect from Oracle:

```sql
EXEC SQL COMMIT WORK RELEASE;
```

The optional keyword WORK provides ANSI compatibility. The RELEASE option frees all Oracle resources (locks and cursors) held by your program and logs off the database.
**Error Handling**

Using the SQL Communications Area (SQLCA): The SQLCA is a data structure. Its components contain error, warning, and status information updated by the database system whenever a SQL statement is executed. Thus, the SQLCA always reflects the outcome of the most recent SQL operation. To determine the outcome, you can check variables in the SQLCA.

To declare the SQLCA, you should copy it into your program with the INCLUDE or #include statement, as follows:

```sql
EXEC SQL INCLUDE SQLCA;
```

or

```sql
#include <sqlca.h>
```

---

**SQLCA Data Structure**

```c
struct sqlc {  
    char sqlcaid[8];
    long sqlabc;
    long sqlcod;
    struct {  
        unsigned short sqlermn;
        char sqlermc[70];  
    } sqlerm;
    char sqlerrp[8];
    long sqlerd[6];
    char sqlwarn[8];
    char sqlext[8];
};
/* ... */
```

---

**SQLCA Fields**

- **sqlcaid**: This string component is initialized to "SQLCA" to identify the SQL Communications Area.
- **sqlabc**: This integer component holds the length, in bytes, of the SQLCA structure.
- **sqlcod**: This integer component holds the status code of the most recently executed SQL statement.
- **sqlerm**: This embedded structure contains two components (later charts).
- **sqlerp**: Reserved for future use.
- **sqlerd**: This array of binary integers has six elements (later charts).
- **sqlwarn**: This array of single characters has eight elements used as warning flags. Oracle sets a flag by assigning to it the character ‘W’ (later charts).
- **sqlext**: Reserved for future use.

---

**SQLCA.SQLCODE Values**

This integer component holds the status code of the most recently executed SQL statement:

- 0: No error.
- >0: Statement executed but exception detected. This occurs when Oracle cannot find a row that meets your WHERE condition or when a SELECT INTO or FETCH returns no rows.
- <0: Oracle did not execute the statement because of an error. When such errors occur, the current transaction should, in most cases, be rolled back.

---

**SQLCA.SQLERRM Values**

This embedded structure contains the following two components:

- **sqlermn**: Length of the message text stored in sqlermc.
- **sqlermc**: Up to 70 characters of the message text corresponding to the error code stored in sqlcod.

---

**SQLCA.SQLERRD Values**

- **sqlerd[0]**: Future use.
- **sqlerd[1]**: Future use.
- **sqlerd[2]**: Numbers of rows processed by the most recent SQL statement.
- **sqlerd[3]**: Future use.
- **sqlerd[4]**: Offset that specifies the character position at which a parse error begins in the most recent SQL statement.
- **sqlerd[5]**: Future use.
**SQLCA.SQLWARN Values**

sqlwarn array of single characters has eight elements used as warning flags. Oracle sets a flag by assigning to it the character 'W'.

- **sqlwarn[0]**: Set if any other flag is set.
- **sqlwarn[1]**: Set if a truncated column value was assigned to an output host variable.
- **sqlwarn[2]**: Set if a NULL column value is not used in computing a SQL aggregate such as AVG or SUM.
- **sqlwarn[3]**: Set if the number of columns in SELECT does not equal the number of host variables specified in INTO.
- **sqlwarn[4]**: Set if every row in a table was processed by an UPDATE or DELETE statement without a WHERE clause.
- **sqlwarn[5]**: Set if a procedure/function/package/package body creation command fails because of a PL/SQL compilation error.
- **sqlwarn[6]**: No longer in use.
- **sqlwarn[7]**: No longer in use.

*SQL from a Programming Language*

---

**WHENEVER Statement**

By default, precompiled programs ignore Oracle error and warning conditions and continue processing if possible. To do automatic condition checking and error handling, you need the WHENEVER statement.

With the WHENEVER statement you can specify actions to be taken when Oracle detects an error, warning condition, or "not found" condition. These actions include continuing with the next statement, calling a routine, branching to a labeled statement, or stopping.

You code the WHENEVER statement using the following syntax:

```sql
EXEC SQL WHENEVER <condition> <action>;
```

*SQL from a Programming Language*

---

**<Conditions>**

You can have Oracle automatically check the SQLCA for any of the following conditions.

- **SQLWARNING**: `sqlwarn[0]` is set because Oracle returned a warning. For example, it is set when Oracle assigns a truncated column value to an output host variable.
- **SQLERROR**: `sqlcode` is set to negative because Oracle returned an error.
- **NOT FOUND**: `sqlcode` is set to positive because Oracle could not find a row that meets your WHERE-clause search condition, or a SELECT INTO or FETCH returned no rows.

*SQL from a Programming Language*

---

**<Actions>**

When Oracle detects one of the preceding conditions, you can have your program take any of the following actions.

- **CONTINUE**: Your program continues to run with the next statement if possible.
- **DO**: Your program transfers control to an error handling function in the program.
- **DO BREAK**: An actual "break" statement is placed in your program. Use this action in loops. When the WHENEVER condition is met, your program exits the loop it is inside.
- **DO CONTINUE**: An actual "continue" statement is placed in your program. Use this action in loops. When the WHENEVER condition is met, your program continues with the next iteration of the loop it is inside.
- **GOTO label_name**: Your program branches to a labeled statement.
- **STOP**: Your program stops running and uncommitted work is rolled back.

*SQL from a Programming Language*

---

**Example 7 - Error Handling**

If you want your program to:
- go to `close_cursor` if a "no data found" condition occurs,
- continue with the next statement if a warning occurs, and
- go to `error_handler` if an error occurs

simply code the following WHENEVER statements before the first executable SQL statement:

```sql
EXEC SQL WHENEVER NOT FOUND GOTO close_cursor;
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER SQLERROR GOTO error_handler;
EXEC SQL WHENEVER SQLWARNING DO print_warning_msg(1);
```

*SQL from a Programming Language*

---

Here is a more concrete example:

```c
/* code to find employee name given employee number */
/* ... */
for (j=1) {
  printf("Give employee number : ");
  scanf("%d", &enum);
  EXEC SQL WHENEVER NOT FOUND GOTO notfound;
  EXEC SQL SELECT ename INTO :emp_name
      FROM employee
      WHERE empno = :enum;
  printf("Name of employee is %s\n", emp_name);
  continue;
notfound:
  printf("No record exists for employee number %d\n",
         enum);
} /* ... */
```

*SQL from a Programming Language*
Example 8 - A Sample Program

```c
#include <stdio.h>
#define TRUE 1
#define FALSE 0

/* Data declare section. Define host variable */
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR uid[20];
  VARCHAR pwd[20];
  VARCHAR user_password[1];
  CHAR sql_stmt[256];
  VARCHAR user_id[10], password[15];
  INT cn num, cn numI N;
  CHAR cn name[20];
  CHAR cn city[20];
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
void sql_error(char msg[]); int valid;
```

Example 8  Cont'd

```c
main()
{
  INT log in ok = FALSE, login tries = 0;
  do
  /* Input user id and passwd. In this sample userid and passwd are set as NULL */
  strcpy(user_id.arr, " ");
  user_id.len = strlen(user_id.arr);
  strcpy(password.arr, " ");
  password.len = strlen(password.arr);
  /* connect to oracle */
  EXEC SQL connect :uid identified by :pwd;
  /* Check whether the connection succeeds */
  if (sqlca.sqlcode == 0)
    log in ok = TRUE;
  else
    printf("Connect Failed!");
    login++; // login (1) & (logincount <3),
  /* try to connect three times if the connection fails */
  if (logincount == 3) & (log in ok) // print(""Too many tries at signing on!");
    exit(0);
  while ( ( ! login ok) & ( login tries <3)) ;
  /* Try to connect three times if the connection fails */
  if ( (login tries = = 3) & ( ! login ok))
    printf("Too many tries at signing on!");
    exit(0);
}
```

Example 8  Cont'd

```c
/* Input the customer # */
printf("Enter Customer number: ");
valid = scanf("%d", & cn num I N);
if (valid != 0)
  {
    printf("Invalid input. Please input number
");
    exit(-1);
  }
else
  printf("Customer # is %d
" , cn num I N);
/* Define cursor if you want to return more than one record from query */
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT cn name, cn city
  FROM x customers
  WHERE cn no > :cn num I N
  ORDER BY cn city;
/* Open the cursor */
EXEC SQL OPEN C1;
/* Loop until print all the result records in the cursor */
for ( ; ; )
  {
    EXEC SQL FETCH C1 INTO :numb, :name, :city;
    printf("%d\%s\%20s\%20s\n", numb, name, city);
  }
/* Close the cursor */
EXEC SQL close C1;
EXEC SQL COMMIT WORK RELEASE; /* Log off database */
printf("Disconnected from ORACLE user "/uid.arr); exit(0);
```

Example 8  Cont'd

```c
/* Input the customer # */
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR uid[20];
  VARCHAR pwd[20];
  VARCHAR user_password[1];
  CHAR sql_stmt[256];
  VARCHAR user_id[10], password[15];
  INT cn num, cn numI N;
  CHAR cn name[20];
  CHAR cn city[20];
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;
void sql_error(char msg[]); int valid;
```

Example 8  Cont'd

```c
EXEC SQL WHENEVER SQLERROR CONTINUE;
/* Print error message */
printf("%s\n", msg);
buf_len = sizeof (msg);
sql_strerror(msg, &buf_len, &msg_len);
printf("%s", msg,buf_len, err_msg);
ex (1);
```