CSC 540 Database Course Project
Bookstore Management System

Shreyas Yagna
Vinoth Kumar Chandra Mohan
Harshavardhan Reddy Malipatel
Chao Li

Answer to question 3

The two examples we have use transactions

Example 1: from the class Contractor.java, method addVendor

```
public boolean addVendor(String name, String address, long
phoneNumber,
                  Date contractStartDate, Date contractEndDate, long
bankAccountNumber)
                  throws CustomDBMSException {

            Connection dbConnection = dbConnectionUtil.getConnection();
            CallableStatement insertVendorsStatement = null;
            boolean success = true;

            logger.debug("Adding a new row in the vendors table with
the name "
                        + name + " and phone number " + phoneNumber);

            try {
                  // Get the prepared statement to call the procedure
                  insertVendorsStatement = dbConnection
                              .prepareCall(addVendorQueryString);

                  // Set the parameters
                  insertVendorsStatement.setString(1, name);
                  insertVendorsStatement.setString(2, address);
                  insertVendorsStatement.setLong(3, phoneNumber);
                  insertVendorsStatement.setDate(4, contractStartDate);
                  insertVendorsStatement.setDate(5, contractEndDate);
                  insertVendorsStatement.setLong(6, bankAccountNumber);

                  insertVendorsStatement.execute();

                  // Verify if the passed bank account exists
                  if (DBMSUtils.verifyBankAccount(bankAccountNumber)) {
                        // The passed bank account is valid. Commit the
transaction
                        logger.debug("Vendor succesfully added");
                        dbConnection.commit();
                  } else {
                        // The passed bank account is invalid. Rollback
the transaction
                        logger.warn("The credit card verification
failed. Rolling back...");
                        dbConnection.rollback();
                        success = false;
                  }

                  dbConnectionUtil.closeConnection(dbConnection);

            } catch (SQLException e) {
                  throw new CustomDBMSException("Got an sql 
exception...", e);
            }

            return success;

      }
```

example 2: from the class salePerson.java, method registerCustomer

```
public boolean registerCustomer(long ssn, String gender, Date dob,
            String name, long phone, String emailId, String address,
            long creditCardNo) throws CustomDBMSException {
      // status of the operation
      Connection dbConnection = dbConnectionUtil.getConnection();
      CallableStatement insertCustomerStatement = null;
      boolean success = true;

      logger.debug("Adding a new row in the customers table with the 
name "
                  + name + " and phone number " + phone);

      try {
            // Get the prepared statement to call the procedure
            insertCustomerStatement = dbConnection
                        .prepareCall(registerCustomerQuery);

            // Set the parameters
            insertCustomerStatement.setLong(1, ssn);
            insertCustomerStatement.setString(2, gender);
            insertCustomerStatement.setDate(3, dob);
            insertCustomerStatement.setString(4, name);
            insertCustomerStatement.setLong(5, phone);
            insertCustomerStatement.setString(6, emailId);
            insertCustomerStatement.setString(7, address);
            insertCustomerStatement.setLong(8, creditCardNo);

            insertCustomerStatement.execute();

            // TODO: what to verify here? Verify if the passed bank 
account
            // exists
            if (DBMSUtils.verifyCreditCard(creditCardNo)) {
                  // The passed bank account is valid. Commit the 
transaction
                  logger.debug("Customer " + name
                        + " details successfully updated");
                dbConnection.commit();
            } else {
                // The passed bank account is invalid. Rollback the 
transaction
                // TODO: change the verification statement
                logger.warn("verification failed Rolling back...");
                dbConnection.rollback();
                success = false;
            }

            dbConnectionUtil.closeConnection(dbConnection);

        } catch (SQLException e) {
            throw new CustomDBMSException("Got an sql exception...", 
e);
        }

        return success;
        }
```

Answer to question 4

Our application has architecture based on from down to top procedure architecture, and role based JAVA architecture. The down to top procedure architecture have reduced the complexity of functionality design, and also reduced the unnecessary network traffic and communication. The role based Java application made functions modulate, and present good design pattern.

1 SQL procedures

SQL procedures are compiled SQL file that could be invoked and executed in the database, it helps to reduce the complexity of functional design and implementation from JAVA side, and made SQL queries cluster based on their functions. For example, to insert a new staff column, there are several tables that requires to be modified, staff table which holds all detailed information about this staff, like name, DOB, phone number, and so on. But this staff works in certain department, has a job title and managed by manager or CEO. To correctly represent relationships with other table, and possibly foreign key reference, more than one INSERT INTO queries are required for this insert staff function. We have defined procedures like:

```
create or replace
procedure insert_staff(name_in IN department.name%TYPE)
as
BEGIN
        …//perform multiple queries
END insert_dept;
/
```

So after Java has made a connection to the database, all the Java function need to do is to invoke procedures that have defined and compiled, pass variables to only one SQL statement, instead of perform multiple queries and send variables repeatedly into database, cause unnecessary network traffic.

We have defined insert, update and delete procedures on table:
Customer,
customer_billing_account,
department,
merchandise,
merchandise_stock,
order,
orderdetails,
place_order,
staff,
store,
vendorpurchases,
vendors

We have also defined generate procedures for:
Customer purchase history
Customer total bill

Sales assist
Staff with department
Vendors
Total vendor bill

Here is one example that elaborate how procedure works

Procedure Insert_staff:
First we performs a insert queries into staff table to create one record of staff, including information of SSN, name, job title, gender, DOB, address, phone number and so on. Since this staff works in a certain department, we have to create another relationship of members of between department and staff. We insert the one-to-one relationship staff_id – department id. Since we have more than one store, this staff also should works in one of the stores, we have to create works_in relation between staff_id and store_id. Last but not the least, this staff should be managed by a manager(if this staff is a manager himself then he should directly report to CEO).

Procedure Update_staff:
To update a staff, we need to refer the relationship among different tables using the staff id. So first we select from a staff table where the staff id = staff_id_passed_in, then we update all data fields in this staff record. At the same time, we also need to update members_of table, works_in table, and managed_by table to update the new relation, all relations can be referred by staff_id

Procedure Delete_staff , Delete_staff by name:
We have offered two ways to delete a staff, by his staff_id, or by name. We have also defined rules and constrains in the create table files when we creating the staff table, so the database will know what kind of delete is legal and what is not. We can perform certain checks to see if this staff can be safely deleted, then we cascading delete all related records in other tables

2 From Java application side
In Java application, we have divided all functionality based on roles. Since different roles have different requirement of functionality and authorization. We have created different classes for CEO, manager, merchandise stocker, sales person and so on. For example, a manager should have the authority to add a new staff, delete staff, and update staff information. The manager

3 Design pattern
We have use singleton design pattern for the connection class. In our application, one connection and only one connection should be exists all the time that we have database operations, this design can guarantee absolute consistency of the database

4 Team Role

All team members plays all the roles in the project development, and all contribution is 25% each.