WolfVilla Hotels
Database Management System

CSC 440 Database Management Systems

Project Report #3

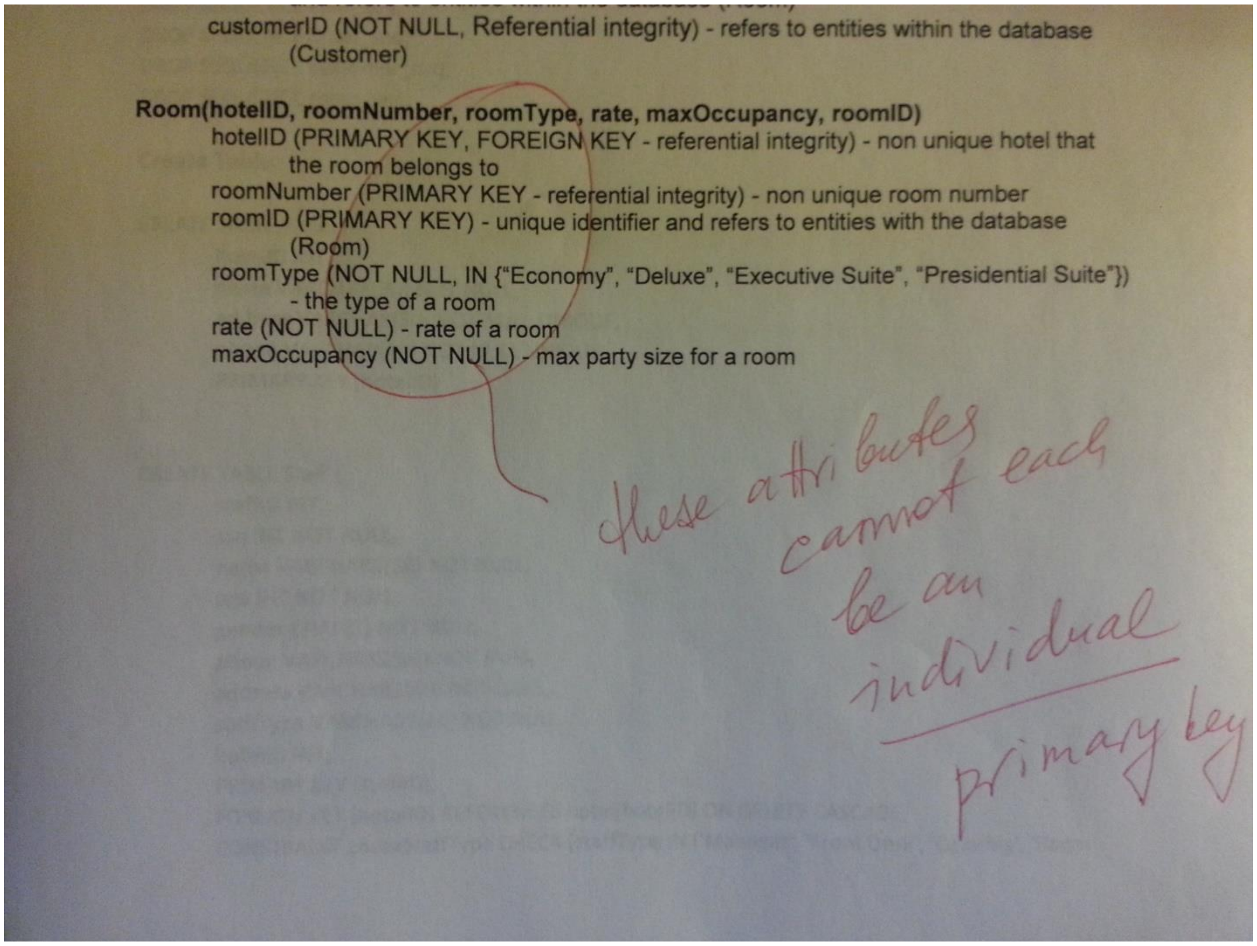
Andrew Poe, Felix Kim, John Hutcherson, Guoyang Di

Assumptions:

1. **A presidential suite has dedicated room service and catering staff as required, but other room types can also have them if the customer opts in at check in.**
2. **The phone, laundry, and restaurant bills are treated as tabs, which the customer can accrue expenses in over their visit, while room service and catering are treated as services that are either provided for the entire visit, or not, and as such are opted in at check in time.**
3. **A staff member works at exactly one hotel.**
4. **The hotel name is not necessarily unique among different hotels, hence the need for a "Hotel ID" key attribute.**
5. **Catering and room service are two separate services.**
6. **The "Staff ID" is unique for every staff member throughout the entire hotel chain, assigned by a global authority for the entire hotel chain.**
7. **Similarly, the global authority assigns each customer a unique ID and every hotel room throughout the entire hotel chain has a unique room ID.**
8. **A customer is able to check into as many hotel rooms as he/she wants. At check-in, the customer says how many people they wish to check into a room (but a room has at most one listed customer). This gives the system the ability to handle situations where a customer has more than 4 people in their party.**
9. **A specific room may only have one customer checked in at a time, that is, the check in table only keeps track of currently checked in guests, not those who used to be checked in, or will be checking in in the future.**
10. **Pricing is static year round. We do not deal with such things as seasonality or day of the week when dealing with pricing.**
11. **Room pricing is determined on a case by case basis, allowing a hotel to have different room rates even among rooms with the same type. This allows more freedom to price more desirable rooms of similar types higher than less desirable ones. Rooms with an ocean view are an example of this.**
12. **Email addresses may be shareable, e.g. spouses or business team members may share a single email address in their communications with the hotels.**
13. **For the purposes of billing, there is no distinction made between credit and debit cards.**
14. **A room has at most one staff member from room service assigned to it. As well as, at most one catering staff assigned to it.**
15. **A room is only present in the CheckIn table during the time period that a check in for the room is valid. Also, when a customer checks into a room, their information is added to the assigned CheckIn table, and when they check out, their information is deleted from the CheckIn table.**
16. **Each hotel has its own unique phone number and unique physical address.**
17. **Any bill cannot fall below 0; a credit can only be applied to a customer's account if there is a corresponding charge whose magnitude is greater than or equal to it.**
18. **As there is no authentication system implemented, users are trusted to log in only as themselves.**
19. **There are no half days, all partial days are rounded up.**
20. **The cost for room service and catering is $50/per day/per service**

Correction for project report 2:

We had one noted problem with report 2, dealing with section 2. The following image is from page 7, counting the title page as a page.



customerID (NOT NULL, Referential integrity) - refers to entities within the database (Customer)

Room(hotelID, roomNumber, roomType, rate, maxOccupancy, roomID)

hotelID (PRIMARY KEY, FOREIGN KEY - referential integrity) - non unique hotel that the room belongs to

roomNumber (PRIMARY KEY - referential integrity) - non unique room number

roomID (PRIMARY KEY) - unique identifier and refers to entities with the database (Room)

roomType (NOT NULL, IN {"Economy", "Deluxe", "Executive Suite", "Presidential Suite"}) - the type of a room

rate (NOT NULL) - rate of a room

maxOccupancy (NOT NULL) - max party size for a room

these attributes cannot each be an individual primary key

The corrected version:

Room(hotelID, roomNumber, roomType, rate, maxOccupancy, roomID)

hotelID (FOREIGN KEY – referential integrity) – many rooms can belong to a hotel

roomNumber (NOT NULL) – non unique room number

roomID (PRIMARY KEY) – unique room identifier among all hotels

roomType (NOT NULL, IN {"Economy", "Deluxe", "Executive Suite", "Presidential Suite"})

- The type of room

rate (NOT NULL, CHECK (rate > 0)) – The cost of the room per night

maxOccupancy (NOT NULL, CHECK (maxOccupancy BETWEEN 1 AND 4)) – max party size allowed to stay in a room

Transaction Code and Documentation:

Transactions were used when adding, editing, and deleting a check-in. When doing one of these actions, the two service staff assignment tables could need to be modified, so it made sense to have a transaction for check-ins. For example, if a service staff was assigned to a room, but then the actual check-in add operation failed, you wouldn't want to leave the service staff assigned to an empty room. Scenarios like this are also applicable to editing and deleting a check-in.

The add, edit, and delete methods are displayed below, followed by the assignRoomServiceStaff, and deleteAssignedRoomServiceStaff methods. The catering equivalents for these last two methods are identical, minus the references to catering instead of room service, and were therefore left off the hardcopy.

```
    @Override
    public boolean add(Connection connection, CheckIn addObj) throws SQLException {
        try {
            conn = DBConnection.getInstance().getDBConnection();
        } catch (Exception e) {
            e.printStackTrace();
        }
        boolean retVal = false;
        PreparedStatement pstring = null;
        try {
            //When we add a check in, we might need to also add an entry to the two
assignedTo
            //tables if the checkin has room service or catering selected.
            conn.setAutoCommit(false);
            conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
            sqlValidator.validate(addObj);

            //Now that the checkin is validated, determine whether to create
assignedTo objects
            if(addObj.getHasCatering().equals("T")) {
                assignCateringStaff(conn, pstring, addObj);
            }
            if(addObj.getHasRoomService().equals("T")) {
                assignRoomServiceStaff(conn, pstring, addObj);
            }

            int results;
            pstring = sqlLoader.loadParameters(conn, pstring, addObj, true);
            results = pstring.executeUpdate();
            conn.commit(); //Commit the changes
            retVal = (results > 0);
            return retVal;
        } catch (SQLException e) {
            //If an SQLException is thrown during the transaction, roll the
            //transaction back to the original state and pass the exception back.
            conn.rollback();
            throw e;
        }finally {
            DBConnection.getInstance().closeConnection(conn, pstring);
        }
    }


@Override
    public boolean update(Connection connection, CheckIn updateObj) throws
SQLException {
        try {
            conn = DBConnection.getInstance().getDBConnection();
        } catch (Exception e) {
            e.printStackTrace();
        }
        boolean retVal = false;
        PreparedStatement pstring = null;
        try {
            conn.setAutoCommit(false);
            conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
            sqlValidator.validate(updateObj);

            //Now that the checkin is validated, determine whether to create
assignedTo objects
            //If the checkin already has staff assigned when it's supposed to, don't
do anything.
            //If staff are supposed to be assigned, but aren't, then assign them.
            //If staff are not supposed to be assigned, attempt to delete any
outstanding assignments.
            if(updateObj.getHasRoomService().equals("T")) {
                RSAssignedTo rs =
rsController.getSpecificRSAssignedTo(updateObj.getRoomID());
                if(rs == null) {
                    assignRoomServiceStaff(conn, pstring, updateObj);
                }
            } else {
                deleteAssignedRoomService(conn, pstring, updateObj);
            }
            if(updateObj.getHasCatering().equals("T")) {
                CAssignedTo c =
cController.getSpecificCAssignedTo(updateObj.getRoomID());
                if(c == null) {
                    assignCateringStaff(conn, pstring, updateObj);
                }
            } else {
                deleteAssignedCatering(conn, pstring, updateObj);
            }

            int results;
            pstring = sqlLoader.loadParameters(conn, pstring, updateObj, false);
            results = pstring.executeUpdate();
            conn.commit();
            retVal = (results > 0);
            return retVal;
        } catch (SQLException e) {
            //If an SQLException is thrown during the transaction, roll the
            //transaction back to the original state and pass the exception back.
            conn.rollback();
            throw e;
        } finally {
            DBConnection.getInstance().closeConnection(conn, pstring);
        }
    }

    @Override
    public boolean delete(Connection connection, CheckIn deleteObj) throws
SQLException {
        try {
            conn = DBConnection.getInstance().getDBConnection();
        } catch (Exception e) {
            e.printStackTrace();
        }
        boolean retVal = false;
        PreparedStatement pstring = null;
        int results;
        try {
            conn.setAutoCommit(false);
            conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

            //Try to delete any outstanding staff assignments
            deleteAssignedCatering(conn, pstring, deleteObj);
            deleteAssignedRoomService(conn, pstring, deleteObj);

            pstring = conn.prepareStatement("DELETE FROM checkin WHERE roomID=?");
            pstring.setInt(1, deleteObj.getRoomID());
            results = pstring.executeUpdate();
            conn.commit();
            retVal = (results > 0);
            return retVal;
        } catch (SQLException e) {
            //If an SQLException is thrown during the transaction, roll the
            //transaction back to the original state and pass the exception back.
            conn.rollback();
            throw e;
        } finally {
            DBConnection.getInstance().closeConnection(conn, pstring);
        }
    }

/**
     * If a checkin requires a room service staff to be assigned, this method
automatically
     * finds the room service staff with the fewest assigned rooms at the hotel where
the
     * room being checked into is located, and then assigned that staff member to the
room.
     * @param connection
     * @param pstring
     * @param addObj
     * @throws SQLException
     */
    private void assignRoomServiceStaff(Connection connection, PreparedStatement
pstring, CheckIn addObj) throws SQLException {
        Room r = roomController.getSpecificRoom(addObj.getRoomID());
        int hotelID = r.getHotelID();
        List<Staff> initialList = staffController.getAllByStaffType("Room Service
Staff");
        List<Staff> specificHotelStaffList = new ArrayList<Staff>();
        //Go through the list, only keeping the staff from the desired hotel
        for(Staff s : initialList) {
            if(s.getHotelID() == hotelID) {
                specificHotelStaffList.add(s);
            }
        }

        //Find only the Assigned entries originating from the desired hotel.
        List<RSAssignedTo> initialAssignedList = rsController.getAll();
        List<RSAssignedTo> specificHotelAssignedList = new ArrayList<RSAssignedTo>();
        for(RSAssignedTo rs : initialAssignedList) {
            if(rs.getHotelID() == hotelID) {
                specificHotelAssignedList.add(rs);
            }
        }

        //Find out which staff member has the fewest assigned jobs, and assign the
job to him/her
        int minCount = -1;
        int rsID = -1;
        for(Staff s : specificHotelStaffList) {
            int count = 0;
            for(RSAssignedTo rs : specificHotelAssignedList) {
                if(rs.getStaffID() == s.getStaffID()) {
                    count++;
                }
            }
            if(minCount == -1) {
                minCount = count;
                rsID = s.getStaffID();
            } else if(count < minCount){
                minCount = count;
                rsID = s.getStaffID();
            }
        }
        if(rsID == -1) {
            throw new SQLException("No available Room Service Staff to assign to
room");
        } else {
            RSAssignedTo rs = new RSAssignedTo(rsID, addObj.getRoomID(),
addObj.getCustomerID(), hotelID);
            rsController.add(connection, rs);
        }
    }



/**
     * Delete the entry from the service assignment table with the roomID equal to
the check-in's roomID
```

```
     * @param connection
     * @param pstring
     * @param obj
     * @throws SQLException
     */
    private void deleteAssignedRoomService(Connection connection, PreparedStatement
pstring, CheckIn obj) throws SQLException {
        String stmt = "DELETE FROM RSAssignedTo WHERE roomID=?";
        pstring = connection.prepareStatement(stmt);
        pstring.setInt(1, obj.getRoomID());
        pstring.executeUpdate();
    }
```

Design Overview:

For the software design there were several important design decisions:

- The connection to the Oracle database server was centralized in a utility class. This allowed any method needing access to the database to just call this class, removing the need for many classes to know how to connect to the DB.
- A script class was written that could automate running our drop, create, and load SQL files. This lets us fully initialize the database by running a single file.
- The cardNumber table from the first two reports was eliminated. Now, if a card is used, the card number is directly stored in the paymentType attribute. This results in a much faster checkin operation, as less tables need to be modified during the operation. It also makes validation easier.
- The overall structure of the program is broken down as follows:
    - Each entity has the following:
        - A data entity class, which only containes data + getters/setters
        - A SQL class, which handles the actual SQL statements, queries, and updates required.
        - A data interface, which defines the methods in the SQL class
        - A SQL Loader class, which takes care of loading prepared statements and interpreting ResultSets.
        - A Validator, which validates the entity before it is added to the database.
        - And a controller, which compiles the functionality of the entity into a single location.
    - A central controller has instances of each entity controller, and contains methods for all functionality the UI needs using any of those controllers.
    - The UI class uses the central controller as a single point of access for the rest of the back end of the program.

Functional Roles:

- Software Engineer – Prime: Andrew, Backup: John
- Database Admin – Prime: John, Backup: Felix
- Test Plan Engineer – Prime: Felix, Backup: Di
- Application Programmer – Prime: Di, Backup: Andrew

At least in theory these were the roles. Really though, everyone did what was needed at the time, so everyone did a good chunk of every role.