WolfPubDb

WolfCity Publishing House Management System

CSC 540 Database Management Systems

Project Report 3

Team L

Akash Srikanth (asrikan2)

Rachit Shah (rshah25)

Shraddha Dhyade (sddhyade)

Venkata Santosh Pavan (vpisapa)

Assumptions

1) Journals and Magazines do not have an ISBN assigned to them.
2) Each Edition of a book has a unique ISBN globally.
3) The Publication Manager assigns the editor(s) to a Publication. However, we do not store who assigned the editor(s) to a Publication.
4) All the Chapters in a book are assumed to have the same set of Authors unlike in a Journal where it is assumed each Article can be written by any set of Authors.
5) Authors and Journalists are the same as the project narrative does not mention any significant differences.
6) Authors do not use the database management system. They only collect payments when they are generated.
7) If an employee comes back after the employment is terminated, the staffID changes and does not remain the same.
8) Authors are assigned to a particular Edition of a Book since different Editions of the same Book might have different sets of Authors.
9) The Administrator and the Billing team is assumed to be the same entity. The reports are also assumed to be generated by the Administrators.
10) The Publication Manager enters the payment data into the details in the payment table and the billing staff uses this to generate the payment. After the generation it is claimed.
11) Publication Manager is responsible for two tasks, publishing and production.
12) An Order corresponds to only one Distributor. If a Publication needs to be assigned to multiple Distributors, each of them will have a different orderID.
13) Each article in an issue can have different topics while all chapters in a book have the same topic so we don't have topic attributes for chapters. We assume that we have a single topic for an entire book and no single topic for Journals or Magazines.
14) Total revenue, total expenses, no. of distributors are calculated since the start of publishing house.
15) No two users can have the same username. If an employee comes back after the employment is terminated, a different username is created.
16) A key with multiple attributes is represented using curly braces. Multiple single attribute keys are mentioned as comma separated values.
17) Two or more Articles can have the same title.
18) Two or more Chapters can have the same title.
19) One contactPerson can be associated with only one Distributor. A distributor can provide exactly one contactPerson.
20) One street address and city can have only one distributor at the most.
21) One phone number cannot serve two different distributors.
22) All staff members must have a joinDate.
23) For an Issue, the date of issue is the same as the publication date.
24) We have assumed that all primary keys will have to be not-null.

Changes to schema from previous reports:

Changes from Report 1 to Report 2

The following changes were made to the schema from report 1's E-R Diagrams. The changes already have the normal forms and integrity constraints proved in report 2.

1) We have changed the Payment Table (pgno: 4) to remove recipientID and recipientType (staff or distributor) which denoted which table the id is from. We did this because the tables GetsPaid connects the Staff and Payment, and DistGetsPayment connects the Distributor and Payment anyway. Adding recipientID will cause redundancy, so we removed it.
2) We have changed the Staff, Editors and Authors table. We removed type from the Staff table from our previous report and added it to Editor and Author instead since type attribute ('internal' and 'external') is only relevant to Editors and Authors.

Changes from Report 2 to Report 3 (Demo)

The following changes were made to the schema and APIs from report 2:

1) **Added 3 additional tables** to the schema (Bill, DistGetsBill, BillConnectedToOrder). These tables are used to store all the bills for each order of the distributor. They are populated when an order is inserted in the DB and a bill is generated for the order to update the outstanding balance of the distributor. Bill table stores the amount owed and bill details. DistGetsBill associates the bill with the distributor and BillConnectedToOrder associates the bill to the order. The normal forms and integrity constraints are given below:
    a) **Bill (<u>billID</u>, billAmt, billDetails)**
    Keys: billID
    FDs: billID -> billAmt, billDetails
    Since LHS is a superkey in this relation we can conclude, it is already in both BCNF and 3NF.
    Primary Key - billID (int)
    UNIQUE - None
    NOT NULL - billAmt because each bill must have an amount associated with it when created. billDetails because it needs to contain bill information to be viewed later on.
    NULL allowed - None
    Foreign Key (Referential Integrity) - None

    b) **DistGetsBill (<u>distID, billID</u>)**
    Keys: {distID, payID}

FDs: distID, billID -> distID, billID (trivial)
Since LHS is a superkey in this relation we can conclude, it is already in both BCNF and 3NF.
Primary Key - {distID, billID} is a composite key (int, int)
UNIQUE - None
NOT NULL - None
NULL allowed - None
Foreign Key (Referential Integrity) - distID (int) because it will access the distID in Distributors relation and billID (int) references the billID in Bill relation.

c) **BillConnectedToOrder (billID, orderID)**
Keys: {billID,orderID}
FDs: billID, orderID -> billID, orderID (trivial)
Since LHS is a superkey in this relation we can conclude, it is already in both BCNF and 3NF.
Primary Key - {billID,orderID} is a composite key (int, int)
UNIQUE - None
NOT NULL - None
NULL allowed - None
Foreign Key (Referential Integrity) - billID (int) because it will access the billID in Bill relation and orderID (int) references the orderID in Orders relation.

2) **Added following columns** (All have been added as null allowed columns since they aren't necessary for any operation. None of them have any foreign keys and none of them are keys)
   a) deliveryDate (DATE) to Orders. We only had orderDate before but based on the demo data, we saw deliveryDate was also needed. deliveryDate is added to the FDs from before. No new functional dependency added, so no change in normal forms.
   b) staffName, gender, age, phone, email and address to the Staff relation. Based on the demo data, we saw these fields needed to be added. Phone and email are unique so the following 2 FDs are formed:
      i) phone -> staffID, joinDate,username, password, salary, staffName, gender, age, email, address
      ii) email -> staffID, joinDate,username, password, salary, staffName, gender, age, phone, address

      Both phone and email are superkeys, so the entire Staff relation is still in BCNF and 3NF.
3) **Changed all DATETIME columns to DATE**. We thought before we needed time but based on demo data we see only the date is needed.
4) **Added ON DELETE CASCADE to all foreign keys.** The previous schema had some but some were missing.
5) **Changed functionality of Distribution Task API number 6 (Generate Bill for a Distributor for a particular order) and number 7 (Change outstanding balance of a distributor on receipt of a payment)** - The combined operations are similar but the

time of when they are executed is changed. Before, we inserted into the Payment, DistGetsPayment and PaymentConnectedToOrder in API number 6 of the generating bill. But this was not correct since the Payment table should record the actual payments and not the future expected payments (bills). Hence, this insertion is now done in number 7 API instead where the distributor actually pays the publication house. To maintain bill information in API number 6 though we added the Bill, DistGetsBill and BillConnectedToOrder tables to maintain this information in DB for future viewing. Updating the balance in both number 6 and 7 is still the same.

6) **Changed functionality of Production Task API "Add Chapter Text"**
   This API would previously add a new chapter entry with no text to the Chapter table. It would then assign this Chapter to an Edition and then update the text.
   According to the modified functionality, this API may now be used to append text to the already existing text of a Chapter. We have modified this functionality so as to distinguish it from the "Update Chapter Text" API.
7) **Changed functionality of Production Task API "Add Article Text"**
   This API would previously add a new article entry with no text to the Article table. It would then assign this Article to a Journal or Magazine and then update the text.
   According to the modified functionality, this API may now be used to append text to the already existing text of an Article. We have modified this functionality so as to distinguish it from the "Update Article Text" API.
8) **Added functionalities in Production task for Finding Article by Topic, Date and Author Name**
   These APIs are similar to "Find Books by Topic", "Find Books by Date", and "Find Books by Author Name". They enable us to view the Articles information by searching using different fields.
9) **Changed functionality of Production Task "Find Payment by PayID"**
   This API is enhanced to view the Payment Information as well as Information associated with it. It will display Staff information if the payment is associated with Staff payment. It will display Orders and Distributors Information if the payment is associated with Orders payment.
10) **Changed functionality of Production Task "Find Payment by Claimed Status"**
    This API is enhanced to view the Payment Information as well as Information associated with it. It will display Staff information if the payment is associated with Staff payment. It will display Orders and Distributors Information if the payment is associated with Orders payment.

Transactions

Transaction 1

Located in distribution() function inside the program code. It is operation number 7 (case 7 of switch case block in the function). Starts at line and ends at line 3655 of Main.java.

Code

```
case 7:
System.out.println("------------------------------------------------------
----------------------");
System.out.println("==> Change outstanding balance of a distributor on
receipt of a payment <==");
System.out.println("------------------------------------------------------
----------------------");
System.out.println("Enter Order ID connected to payment: ");
ordID = sc.nextInt();
//check whether the order exists
query = "SELECT * FROM Orders WHERE orderID="+ordID;
try{
  ResultSet rs = dbManager.executeQuery(query);
  if(!rs.next()){
    System.out.println("The provided order ID doesn't exist in the DB");
    break;
  }
}
catch(Exception e){
  e.printStackTrace();
  System.out.println("Error searching order ID in DB");
  break;
}
System.out.println("Enter Amount of Payment: ");
double payAmt = sc.nextDouble();
```

```
query = "SELECT distID,name,city FROM DistAssigned NATURAL JOIN
Distributors WHERE orderID="+ordID;
try{
  ResultSet rs = dbManager.executeQuery(query);
  if(!rs.next()){
    System.out.println("The provided order ID doesn't exist in the DB");
    break;
  }
  distID = rs.getInt("distID");
  distName = rs.getString("name");
  distCity = rs.getString("city");
}
catch(Exception e){
  e.printStackTrace();
  System.out.println("Error searching order ID in DB");
  break;
}
System.out.println("The provided order ID corresponds to distributor
"+distName+" (ID:"+distID+") from "+distCity+" city.");

//Get pay ID for Payment
boolean payFlag = true;
Random rand = new Random();
payID = Math.abs(rand.nextInt());
while(payFlag){
  System.out.println("Enter pay ID for the new payment: ");
  payID = sc.nextInt();
  try{
    query = "SELECT payID from Payment WHERE payID="+payID;
    ResultSet rs = dbManager.executeQuery(query);
    if(!rs.next()){
      payFlag = false;
    }
```

```
    else{
      System.out.println("Pay ID already exists. Try again.");
    }
  }
  catch(Exception e){
    System.out.println("Error checking payID. Setting to a random
integer");
    payID = Math.abs(rand.nextInt());
  }
}
sc.nextLine();
//Set Payment Date
System.out.println("Enter date of payment: [YYYY-MM-DD][Leave blank if you
want today's date]:");
String payDate = sc.nextLine();
if(!checkValidDate(payDate)){
  if(payDate.equals("")){
    payDate = LocalDate.now().toString();
  }
  else{
    System.out.println("Invalid Date Format");
  }
}

//Create Payment using transactions to maintain atomicity
try{
  //start transaction (set auto-commit false)
  dbManager.startTransaction();

  //Step 1: Add to Payment table
  query = "INSERT INTO Payment (payID, generatedDate, amount, claimedDate)
VALUES (%d,\"%s\",%f,\"%s\")";
  query = String.format(query,payID,payDate,payAmt,payDate);
  dbManager.executeUpdate(query);
```

```
  //Step 2: Associate the payment with the distributor.
  query = "INSERT INTO DistGetsPayment (payID,distID) VALUES (%d,%d)";
  query = String.format(query,payID,distID);
  dbManager.executeUpdate(query);

  //Step 3: Associate the payment with the order
  query = "INSERT INTO PaymentConnectedToOrder (payID, orderID) VALUES
(%d,%d)";
  query = String.format(query,payID,ordID);
  dbManager.executeUpdate(query);

  //Step 4: Update the outstanding balance of the distributor
  query = "UPDATE Distributors SET balance=balance-"+payAmt+" WHERE
distID="+distID;
  dbManager.executeUpdate(query);

  //If all updates run correctly, commit and set auto-commit back to true
  dbManager.commitTransaction();
  System.out.println("~~~ Successfully updated outstanding balance of
Distributor and added Payment details ~~~");
}
catch(Exception e){
  //If anything fails, rollback and set auto-commit back to true
  try{
    dbManager.rollbackTransaction();
  }
  catch(Exception ex){
    System.out.println("!!! Error rolling back transaction. Connection to
DB Lost. !!!");
  }
  System.out.println("!!! Error in transactions. Rolling back. !!!");
}
break;
```

Documentation

The operation is used to update the outstanding balance of a distributor on receipt of a payment for a particular order. The whole operation can be divided in the following steps:

1) Get payment information (id of payment to be inserted, amount of payment and date of payment) and insert it to the Payment table.
2) Associate the payment to the distributor in the DistGetsPayment table (distID is determined from the DistAssigned table using the orderID which is given by the user)
3) Associate the payment to the order in the PaymentConnectedToOrder table using the same orderID and payID.
4) Update outstanding balance of distributor (balance parameter in Distributors table) using formula balance=balance-payAmt.

All other parts of code are getting input from the user and making sure it is present in the DB. The above 4 steps needs to be done atomically, i.e., If one fails, none of them should pass. For this transactions are used to maintain this atomicity. At the start of the try-catch block, we use our helper class DBManager to start a transaction by setting auo-commit to false. And after executeUpdate of all the above 4 steps, if all succeeds in the try block we commitTransaction() using the DBManager again which essentially calls commit() and sets auto-commit back to true. If anything fails, we rollback to a savepoint before this transaction so that none succeeds. This is done by calling the rollbackTransaction() of DBManager which essentially calls rollback() and sets auto-commit back to true. The transaction is done using the default isolation level of serializable because we don't want updates done by users parallely using this program to reflect to the current user.

Transaction 2

Located in production() function inside the program code. It is operation number 16 (case 16 of Production switch case block in the function). Starts at line 2259 and ends at line 2342 of Main.java.

Code

```
//Add Article to an Issue
   case 16:
  System.out.println("16. Add new Article to an Issue in DB:");
  checkFlag = false;
  try{
  //Take user input for a new Article ID
  System.out.println("Enter Article ID for the Article:");
  artId = sc.nextInt();
  sc.nextLine();
```

```
   //Take user input for a new Article number for the Article in the Issue
   System.out.println("Enter Article number for the Article in the
Issue:");
   artNum = sc.nextInt();
   sc.nextLine();
   //Take user input for Article topic
    System.out.println("Enter topic for Article:");
    artTopic = sc.nextLine();
   //Take user input for Article title
    System.out.println("Enter title for Article:");
    artTitle = sc.nextLine();
   //Take user input for Article text
   System.out.println("If you would like to enter text to this article
please type below. Press Enter to skip:");
   artText = sc.nextLine();
   //Take user input for Article date
   System.out.println("If you would like to enter date for this Article,
please type below [YYYY-MM-DD]. Press Enter to skip:");
   date = sc.nextLine();
   if(date.equals(""))
     date = "NULL";
    else{
   if(!checkValidDate(date)){
               System.out.println("Invalid Date Format.");
               break;
   }
   else
     date = "\"" + date + "\"";
    }
  //Take user input for Publication ID of the Issue to which you want to
assign the Article
   System.out.println("Enter Publication ID of the Issue to which you want
to assign this Article:");
   pubId = sc.nextInt();
 //Take user input for Issue ID of the above Publication
   System.out.println("Enter Issue ID of the Issue to which you want to
assign this Article:");
```

```
    issueId = sc.nextInt();

    sc.nextLine();
   }
     catch(InputMismatchException e){
       e.printStackTrace();
       System.out.println("Input type error!");
       sc.nextLine();
       break;
   }
    query2 = "INSERT INTO Article(artID, topic, title, date, text) VALUES
("+artId+", \""+artTopic+"\", \""+artTitle+"\", "+date+",
\""+artText+"\");";

    query3 = "INSERT INTO IssueContains(pubID, issueID, artID, artNum)
VALUES ("+pubId+","+issueId+","+artId+","+artNum+");";

    try{
     //Begin transaction
         dbManager.startTransaction();
         query1 = "SELECT * FROM Journals UNION SELECT * FROM Magazines
WHERE pubID = "+pubId+";";
     rs = dbManager.executeQuery(query1);
     //Enable check flag to check if publication is of correct
type(Journal/Magazine)
     while(rs.next()){
     checkFlag = true;
     }
     //Proceed execution based on check flag
     if(checkFlag){
         dbManager.executeUpdate(query2);
       dbManager.executeUpdate(query3);
       dbManager.commitTransaction();
       System.out.println("Successfully added Article in DB");
       }
   else
```

```
    System.out.println("Entered publication ID is not a Journal or
Magazine type!");
        }
        catch(Exception e){
          e.printStackTrace();
          System.out.println("Error adding Article in DB!");
          try{
            dbManager.rollbackTransaction();
          }
          catch(Exception ex){
            System.out.println("Error rolling back. Connection to DB
lost.");
          }
        }
        break;
```

Documentation

This transaction refers to the 'Add Article to an Issue' API under Production task.
In this API, we have two sub-tasks:

1) Add a new Article
2) Assign the Article to an Issue

First, we take user input required for adding a new article. Then we ask for Publication ID and IssueID to which we want to assign this Article.
Here, we use a transaction to execute these operations because even if one of the operations fail, we want to roll back any changes made to the database.
Thus, we execute both queries in try catch block with startTransaction() of DBManager setting auto-commit to false and commitTransaction() to commit all changes at end of the block and set auto-commit back to true.
We catch any SQL Exceptions occurred in this block and attempt to roll back transactions to revert any changes made during this process to maintain atomicity. The transaction is done using the default isolation level of serializable because we don't want updates done by users parallely using this program to reflect to the current user.

High-Level Design Decisions

We implement the system as a console-based application with menu based navigation. The system has a main class which allows the user to perform operations by the options given in the menu. The options displayed depend on the user role as the user role determines the limitations over what kind of operations can be performed or not performed. For e.g. publication managers can only access editing, publication and production tasks, and have no use for distribution, reports and personnel tasks. Admin can view everything. The view of our other users like Distribution Managers and Editors is also limited like this.

There is a main menu that is displayed after the user logs in with their credentials. This menu consists of the broader options over which the user would like to perform the operations and they are - Editing and Publishing, Production, Distribution, Reports or Managing Personnel.
When selected, another menu, a sub menu, which is specific to that particular selection is displayed.

The operations specific to each of the options in the main menu have been implemented separately in their respective methods of the Main class. This was done as it would make the code more manageable and robust. Sufficient care is taken to make the code more robust by using input checks (for e.g., when updating or deleting a record by ID where the given ID actually exists, checking valid email or date format, etc.) Try-catch blocks make sure all exceptions are caught for continuous operation of the system.

Our code also contains a helper class called DBManager which is used to handle the queries and updates in the database. In addition to that, it is also used to handle the transactions, to avoid any issues while altering the data in the database.

Functional Roles

Part 1:
Software Engineer: Rachit Shah(Prime), Akash Srikanth(Backup)
Database Designer/Administrator: Akash Srikanth(Prime), Shraddha Dhyade(Backup)
Application Programmer: Shraddha Dhyade(Prime), Venkata Santosh Pavan Pisapati(Backup)
There is no Test Plan Engineer role here as there was no code written for this part.

Part 2:
Software Engineer: Venkata Santosh Pavan Pisapati(Prime), Shraddha Dhyade(Backup)
Database Designer/Administrator: Shraddha Dhyade(Prime), Rachit Shah(Backup)
Application Programmer: Rachit Shah(Prime), Akash Srikanth(Backup)
Test Plan Engineer: Akash Srikanth(Prime), Venkata Santosh Pavan Pisapati(Backup)

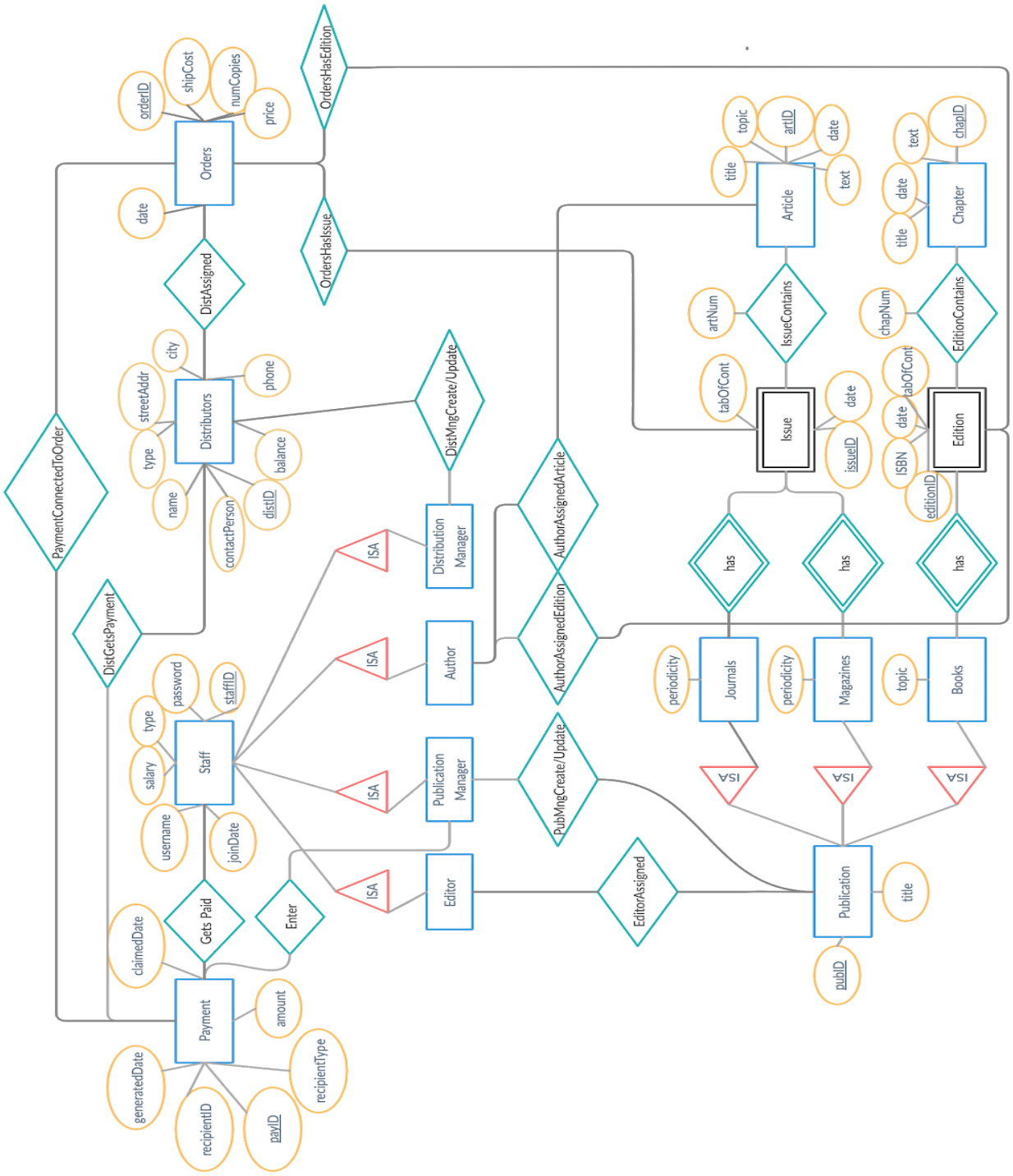
Part 3:
Software Engineer: Shraddha Dhyade(Prime), Rachit Shah (Backup)
Database Designer/Administrator: Venkata Santosh Pavan Pisapati(Prime), Shraddha Dhyade (Backup)
Application Programmer: Akash Srikanth(Prime), Venkata Santosh Pavan Pisapati(Backup)
Test Plan Engineer: Rachit Shah(Prime), Akash Srikanth(Backup)

Revision to Previous Reports

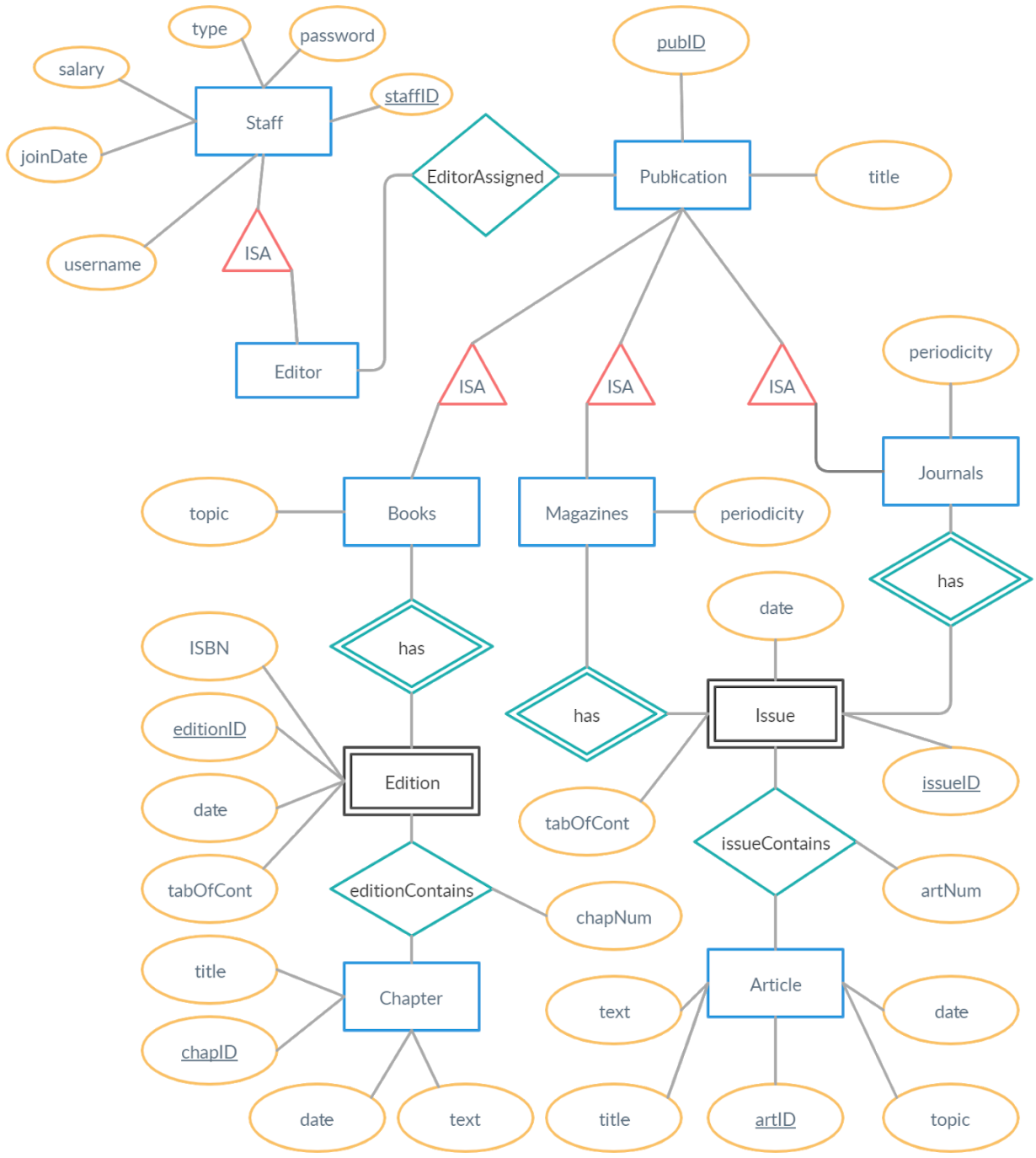
We lost 20 points in report 1 as the clarity of the attached ER diagram images were not clear (because of unclear printing out).
The below are the same images in a better resolution.
We lost no points in report 2, so no changes were made.

Local E/R Diagrams

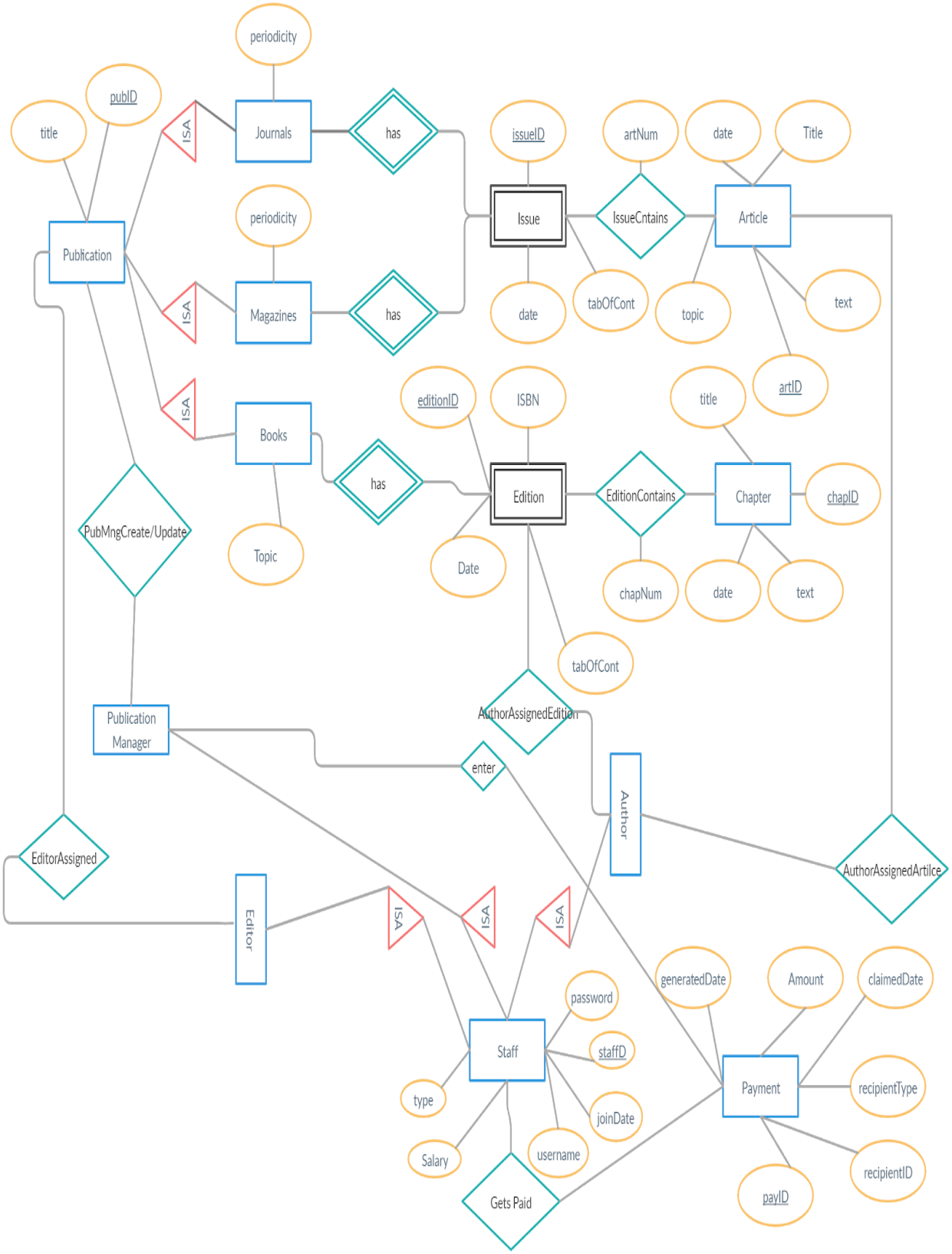
Admin's View:



Editor's View:



Publication Manager :



Distribution Manager's View: