Project DOC

Database Design:

- The relational diagram of our DB shows all the tables with all the attributes and their data types.
- The primary, alternate and foreign keys have been highlighted separately.
- Wherever we have set up a foreign key referencing, we have added on delete cascade & update cascade so as to ensure db consistency.
- The database has been normalized to the BCNF normal form
- Wherever there are pk's / id's we have added the not null constraints and declared that the value be non-negative and be auto-incremented
- The basic layout contains **entities** and liking **relations** (as in diagram), for example:
  Books and authors linked via a "written_by" relation, linking each book to one author
  Book linked to the genre via "book_genre" relation, linking each book to a set of genres via row entries

Constraints:

- *Referential Integrity Constraint*: Wherever we have used Foreign Keys, we have set up the cascade constraints as mentioned above.

- *Domain integrity*: we have ensured through the front end that whenever a user tries to insert a value which does not belong to the domain of that field then we notify the user and stops that transaction at that point and no changes takes place (rollback) also SQL exceptions are also handled so there is no case left through which value out of domain could be inserted in the database also

- *Range values*: We have used the not null, unique, check(> 0) i.e. non-negative value, etc whenever need be, so as to ensure consistency

- Trigger : book_out_of_stock trigger, whenever a book runs out of stock, it gets automatically removed the cart of all users who have that particular book in their cart.

- Procedure :
  BonusInrement() , called from the database backend to give a 10% bonus to all users on account of festivities or at the end of each year or so.

  getAllBooks(), getAllGenre() // helper procs needed for debugging,not used in app

Optimizing querying process:

- Indexing :

  We have indexed attributes where we thought the table needs frequent accesses either directly or via queries, or where the table size only grows.
  We have indexed b_id , b_name from books,
  u_id , username, contact from useraccounts.
  o_id , u_id, b_id from all_order
  We have created (b_id, auth_id, g_id) combined index for the MV.

- Materialized View
  We have created a materialized view for our home table as filling it is an operation with quite some overhead. We can refer to the data for the home table from the view.

Functionalities:

Reader has a :
- Wallet for payment and transfer to friends, transaction history to show the same
- Wishlist for books to read later on
- Cart to buy all at once
- Purchase history to view purchases
- Account profile

Publications can:
- Restock old books
- Bring new books to the store

Authors (spectative stakeholder) can:
- View their bestselling book
- View their best selling genre
- View their average rating as an author
- View how other authors are performing relative to them
- View how each of their books are performing

Stock Manager :
- Has access the entire stock
- Can filter out the stock as per order needs.
-

Company Head:
- Can audit the store’s week wise finances
- Has access to publication wise share of the book stock

GUI classes & functions using JavaFx :

The GUI for the project has been developed using javafx. There we build fxml files which contain the basic view elements , like tables, navigation buttons, labels etc.

The data for the tables and charts, text for the labels etc are picked up from the database. This is done using java functions written inside controller classes.

Each **fxml file** is linked to a **controller file** for carrying out these functionalities, initializing the views, etc.

The functions for accessing db to show data on gui are in these classes.
The functions linked to buttons for update queries are also in these controller classes.

So on and so forth…………………….