MULTIMEDIA  UNIVERSITY ®

CIT6114 Database Fundamentals

Assignment 2

Group 2

Title: SHDL MMU Library System

T10L

Prepared by:

	ID	NAME
Leader	242UC243TL	SITI ZULAIKHA BINTI ABDUL RAZIF
Members	242UC243BE	EBA MOHAMED ABBAS AHMED
	242UC243B4	LAMA M. R. SIAM
	243UC246VX	ESHIKA PRASAAD

Business Rules:

1. A user borrows multiple loans, and each loan contains multiple loan details.
Entities involved: user, loan, loan_detail
Explanation: The loan table connects users to specific loans via user_id, allowing a user to have many loans. The loan_detail table links to loan_id, enabling each loan to record multiple return and renewal instances.

2. A user makes multiple reservations, but each reservation is for one specific book.
Entities involved: user, reservation, book.
Explanation: The reservation links the user and the book, allowing many reservations per user.

3. A book belongs to one genre, but a genre can include multiple books.
Entities involved: book, genre.
Explanation: The foreign key in the book table points to genre_id, defining a one-to-many relationship from genre to book.

4. A book is stored at one branch, but a branch can store multiple books.
Entities involved: book, branch.
Explanation: The book table has a foreign key to branch_id, indicating a many-to-one relationship.

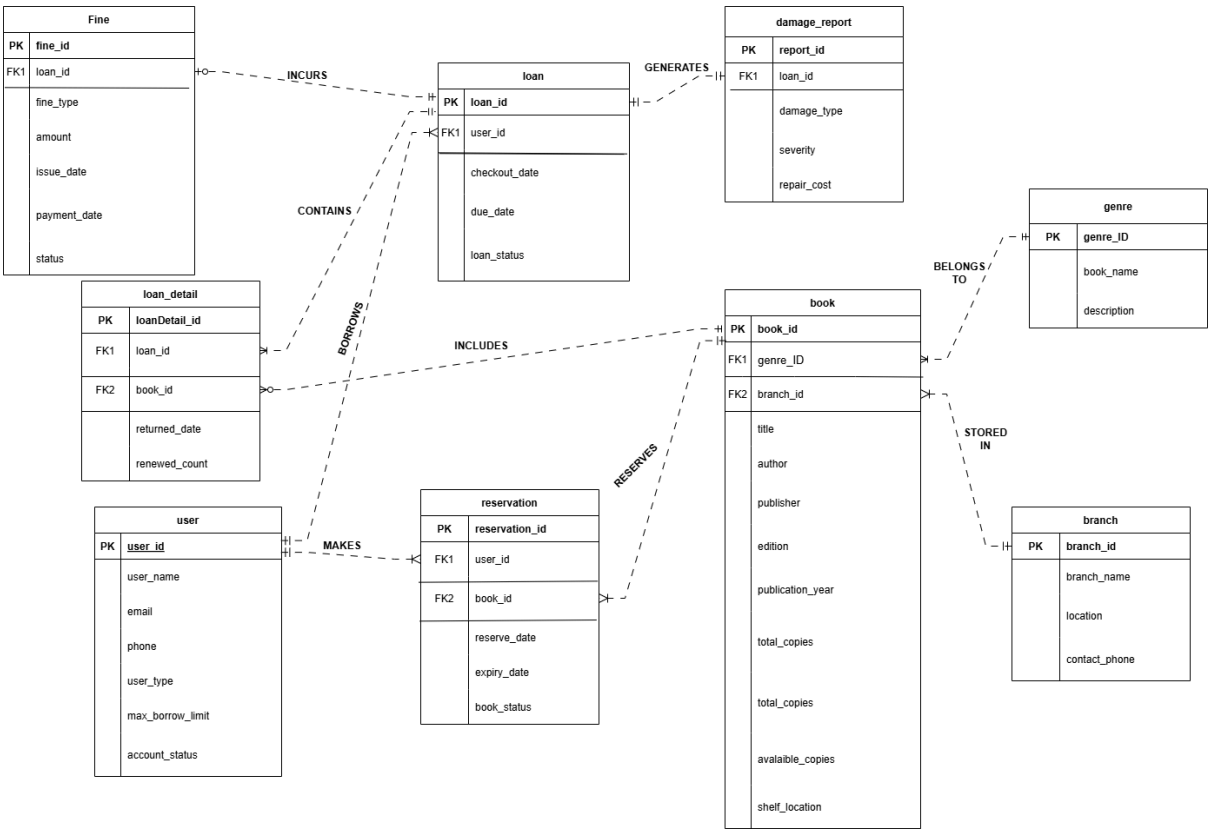
5. A loan may result in a fine, but each fine is linked to a single loan.
Entities involved: fine, loan.
Explanation: The fine table includes foreign key for loan_id, establishing direct connections.

6. Each loan must have exactly one damage report, and each damage report is linked to exactly one loan.
Entities involved: damage_report, loan
Explanation: damage_report has a foreign key to loan_id, implying one report per loan.

7. Users have a maximum borrow limit depending on their user type.
Entities involved: user. Explanation: The attribute max_borrow_limit in the user table enforces borrowing rules tied to user classification.
8. Book availability is tracked by the number of available copies.
Entities involved: book. Explanation: The book table has total_copies and available_copies attributes to manage circulation.

9. Each loan detail is connected to exactly one loan and one book. A loan has many loan details. A book may be linked to multiple loan details.
Entities involved: loan_detail, loan, book
Explanation: The loan_detail table contains foreign keys to both loan_id and book_id. This forms a one-to-many relationship from loan to loan_detail, and an optional many-to-one from loan_detail to book.

ERD:


Fine
PK fine_id
FK1 loan_id
fine_type
amount
issue_date
payment_date
status
INCURS
loan
PK loan_id
FK1 user_id
checkout_date
due_date
loan_status
GENERATES
damage_report
PK report_id
FK1 loan_id
damage_type
severity
repair_cost
genre
PK genre_ID
book_name
description
BELONGS TO
loan_detail
PK loanDetail_id
FK1 loan_id
FK2 book_id
returned_date
renewed_count
CONTAINS
BORROWS
INCLUDES
book
PK book_id
FK1 genre_ID
FK2 branch_id
title
author
publisher
edition
publication_year
total_copies
total_copies
avalaible_copies
shelf_location
STORED IN
RESERVES
user
PK user_id
user_name
email
phone
user_type
max_borrow_limit
account_status
MAKES
reservation
PK reservation_id
FK1 user_id
FK2 book_id
reserve_date
expiry_date
book_status
branch
PK branch_id
branch_name
location
contact_phone

Data Dictionary:

Table Name	Attribute Name	Contents	Type	Required	PK or FK	FK Referenced Table
USER	user_id	User ID	CHAR(5)	Y	PK	
	user_name	User full name	VARCHAR(50)	Y		
	email	User email address	VARCHAR(50)	Y		
	phone	User phone number	CHAR(10)	Y		
	user_type	Type (student/staff)	VARCHAR(10)	Y		
	max_borrow_limit	Max borrowable books	NUMBER(2)	Y		
	account_status	Active/suspended	VARCHAR(10)	Y		
BOOK	book_id	Book ID	CHAR(5)	Y	PK	
	genre_ID	Genre ID	CHAR(3)	Y	FK	GENRE
	branch_id	Branch location ID	CHAR(3)	Y	FK	BRANCH
	title	Book title	VARCHAR(100)	Y		
	author	Book author	VARCHAR(50)	Y		
	publisher	Publisher name	VARCHAR(50)	Y		
	edition	Edition number	VARCHAR(10)	Y		
	publication_year	Year of publication	CHAR(4)	Y		
	total_copies	Number of total copies	NUMBER(3)	Y		
	available_copies	Copies currently available	NUMBER(3)	Y		
	shelf_location	Book shelf location	VARCHAR(20)	Y		
LOAN	loan_id	Loan ID	CHAR(5)	Y	PK	
	user_id	User ID	CHAR(5)	Y	FK	USER
	checkout_date	Book checkout date	DATE	Y		
	due_date	Due date for return	DATE	Y		
	loan_status	Loan status	VARCHAR(20)	Y		
FINE	fine_id	Fine ID	CHAR(5)	Y	PK	
	loan_id	Loan ID	CHAR(5)	Y	FK	LOAN
	fine_type	Type of fine	VARCHAR(20)	Y		
	amount	Fine amount	NUMBER(6,2)	Y		
	issue_date	Date issued	DATE	Y		
	payment_date	Date paid	DATE	N		
	status	Fine status	VARCHAR(20)	Y		
DAMAGE_REPORT	report_id	Report ID	CHAR(5)	Y	PK	
	loan_id	Loan ID	CHAR(5)	Y	FK	LOAN
	damage_type	Type of damage	VARCHAR(50)	Y		
	severity	Severity of damage	VARCHAR(20)	Y		
	repair_cost	Repair cost	NUMBER(6,2)	Y		
LOAN_DETAIL	loanDetail_ID	Loan detail ID	CHAR(5)	Y	PK	
	loan_id	Loan ID	CHAR(5)	Y	FK	LOAN
	book_id	Book ID	CHAR(5)	Y	FK	BOOK
	returned_date	Date book returned	DATE	N		
	renewed_count	Renewed times	NUMBER(2)	Y		
RESERVATION	reservation_id	Reservation ID	CHAR(5)	Y	PK	
	user_id	User ID	CHAR(5)	Y	FK	USER
	book_id	Book ID	CHAR(5)	Y	FK	BOOK
	reserve_date	Date reserved	DATE	Y		
	expiry_date	Reservation expiry	DATE	Y		
	book_status	Book status	VARCHAR(20)	Y		
GENRE	genre_ID	Genre ID	CHAR(3)	Y	PK	
	book_name	Genre name	VARCHAR(30)	Y		
	description	Genre description	VARCHAR(100)	N		
BRANCH	branch_id	Branch ID	CHAR(3)	Y	PK	
	branch_name	Branch name	VARCHAR(50)	Y		
	location	Branch location	VARCHAR(100)	Y		
	contact_phone	Contact number	CHAR(10)	Y		

Table Creation and Insertion:

1. USERS Table

```
CREATE TABLE users (
   user_id CHAR(5) PRIMARY KEY,
   user_name VARCHAR(50) NOT NULL,
   email VARCHAR(50) NOT NULL,
   phone CHAR(10) NOT NULL,
   user_type VARCHAR(10) NOT NULL,
   max_borrow_limit NUMERIC(2) NOT NULL,
   account_status VARCHAR(10) NOT NULL
);

INSERT INTO users VALUES ('U001', 'Alice Smith', 'alice@example.com', '0123456789',
'student', 5, 'active');
INSERT INTO users VALUES ('U002', 'Bob Johnson', 'bob@example.com', '0112233445',
'staff', 3, 'active');
INSERT INTO users VALUES ('U003', 'Charlie Lee', 'charlie@example.com', '0134455667',
'student', 4, 'suspended');
INSERT INTO users VALUES ('U004', 'Diana Wells', 'diana@example.com', '0172233445',
'student', 6, 'active');
INSERT INTO users VALUES ('U005', 'Ethan Moore', 'ethan@example.com', '0198765432',
'staff', 2, 'active');
```

```
tc3l=# select * from users;
 user_id |  user_name  |        email        |   phone    | user_type | max_borrow_limit | account_status
---------+-------------+---------------------+------------+-----------+------------------+----------------
 U001    | Alice Smith | alice@example.com   | 0123456789 | student   |                5 | active
 U002    | Bob Johnson | bob@example.com     | 0112233445 | staff     |                3 | active
 U003    | Charlie Lee | charlie@example.com | 0134455667 | student   |                4 | suspended
 U004    | Diana Wells | diana@example.com   | 0172233445 | student   |                6 | active
 U005    | Ethan Moore | ethan@example.com   | 0198765432 | staff     |                2 | active
(5 rows)
```

2. branch Table

```
CREATE TABLE branch (
   branch_id CHAR(3) PRIMARY KEY,
   branch_name VARCHAR(50) NOT NULL,
   location VARCHAR(100) NOT NULL,
   contact_phone CHAR(10) NOT NULL
);

INSERT INTO branch VALUES ('B01', 'Cyberjaya', 'Cyberjaya, Selangor', '0987654321');
INSERT INTO branch VALUES ('B02', 'Melaka', 'Melaka City', '0987612345');
```

```
tc3l=# select * from branch;
 branch_id | branch_name |      location       | contact_phone
-----------+-------------+---------------------+--------------
 B01       | Cyberjaya   | Cyberjaya, Selangor | 0987654321
 B02       | Melaka      | Melaka City         | 0987612345
(2 rows)
```

3. genre Table

CREATE TABLE genre (
  genre_ID CHAR(3) PRIMARY KEY,
  book_name VARCHAR(30) NOT NULL,
  description VARCHAR(100)
);

INSERT INTO genre VALUES ('G01', 'Engineering', 'Books related to engineering disciplines.');
INSERT INTO genre VALUES ('G02', 'Business', 'Business studies and management.');
INSERT INTO genre VALUES ('G03', 'Marketing', 'Marketing theories and strategies.');
INSERT INTO genre VALUES ('G04', 'Accounting', 'Accounting standards and practices.');
INSERT INTO genre VALUES ('G05', 'IT', 'Information Technology fundamentals.');
INSERT INTO genre VALUES ('G06', 'Computer Science', 'Core concepts in computing and algorithms.');
INSERT INTO genre VALUES ('G07', 'Multimedia', 'Digital arts, animation, and media studies.');
INSERT INTO genre VALUES ('G08', 'Cinematic Arts', 'Film production and theory.');
INSERT INTO genre VALUES ('G09', 'Communication', 'Mass communication and media analysis.');
INSERT INTO genre VALUES ('G10', 'Finance', 'Financial analysis and investment.');
INSERT INTO genre VALUES ('G11', 'Law', 'Legal studies and judicial systems.');

```
tc3l=# select * from genre;
 genre_id |    book_name     |                description
----------+------------------+-------------------------------------------
 G01      | Engineering      | Books related to engineering disciplines.
 G02      | Business         | Business studies and management.
 G03      | Marketing        | Marketing theories and strategies.
 G04      | Accounting       | Accounting standards and practices.
 G05      | IT               | Information Technology fundamentals.
 G06      | Computer Science | Core concepts in computing and algorithms.
 G07      | Multimedia       | Digital arts, animation, and media studies.
 G08      | Cinematic Arts   | Film production and theory.
 G09      | Communication    | Mass communication and media analysis.
 G10      | Finance          | Financial analysis and investment.
 G11      | Law              | Legal studies and judicial systems.
(11 rows)
```

4. book Table

CREATE TABLE book (

```
    book_id CHAR(5) PRIMARY KEY,
    genre_ID CHAR(3) NOT NULL,
    branch_id CHAR(3) NOT NULL,
    title VARCHAR(100) NOT NULL,
    author VARCHAR(50) NOT NULL,
    publisher VARCHAR(50) NOT NULL,
    edition VARCHAR(10) NOT NULL,
    publication_year CHAR(4) NOT NULL,
    total_copies NUMERIC(3) NOT NULL,
    available_copies NUMERIC(3) NOT NULL,
    shelf_location VARCHAR(20) NOT NULL,
    FOREIGN KEY (genre_ID) REFERENCES genre(genre_ID),
    FOREIGN KEY (branch_id) REFERENCES branch(branch_id)
);
```

INSERT INTO book VALUES ('BK01', 'G06', 'B01', 'The Time Machine', 'H.G. Wells', 'Penguin Classics', '1st', '1895', 10, 9, 'A12');
INSERT INTO book VALUES ('BK02', 'G11', 'B01', 'Sherlock Holmes', 'Arthur Conan Doyle', 'Bantam Classics', 'Reprint', '1986', 7, 7, 'B10');
INSERT INTO book VALUES ('BK03', 'G01', 'B02', 'Dune', 'Frank Herbert', 'Ace Books', 'Reissue', '2005', 5, 5, 'C3');
INSERT INTO book VALUES ('BK04', 'G08', 'B01', 'Harry Potter', 'J.K. Rowling', 'Bloomsbury', 'Illustrated', '2015', 12, 10, 'D5');
INSERT INTO book VALUES ('BK05', 'G10', 'B02', 'Sapiens', 'Yuval Noah Harari', 'Harper', '1st', '2011', 8, 8, 'E2');
INSERT INTO book VALUES ('BK06', 'G04', 'B01', 'Financial Accounting', 'Walter Harrison', 'Pearson', '4th', '2017', 6, 6, 'F1');
INSERT INTO book VALUES ('BK07', 'G07', 'B02', 'Multimedia: Making It Work', 'Tay Vaughan', 'McGraw-Hill', '9th', '2014', 5, 5, 'F2');
INSERT INTO book VALUES ('BK08', 'G03', 'B01', 'Principles of Marketing', 'Philip Kotler', 'Pearson', '17th', '2020', 9, 9, 'G1');
INSERT INTO book VALUES ('BK09', 'G09', 'B01', 'Mass Communication', 'Joseph Dominick', 'McGraw-Hill', '12th', '2012', 4, 4, 'G2');
INSERT INTO book VALUES ('BK10', 'G02', 'B02', 'Understanding Business', 'William Nickels', 'McGraw-Hill', '11th', '2016', 7, 7, 'H1');
INSERT INTO book VALUES ('BK11', 'G05', 'B02', 'IT for Management', 'Efraim Turban', 'Wiley', '10th', '2018', 5, 5, 'H2');

```
tc3l=# select * from book;
 book_id | genre_id | branch_id |          title           |       author       |    publisher     | edition | publication_year | total_copies | available_copies | shelf_location
---------+----------+-----------+--------------------------+--------------------+------------------+---------+------------------+--------------+------------------+----------------
 BK01    | G06      | B01       | The Time Machine         | H.G. Wells         | Penguin Classics | 1st     | 1895             |           10 |                9 | A12
 BK02    | G11      | B01       | Sherlock Holmes          | Arthur Conan Doyle | Bantam Classics  | Reprint | 1986             |            7 |                7 | B10
 BK03    | G01      | B02       | Dune                     | Frank Herbert      | Ace Books        | Reissue | 2005             |            5 |                5 | C3
 BK05    | G10      | B02       | Sapiens                  | Yuval Noah Harari  | Harper           | 1st     | 2011             |            8 |                8 | E2
 BK06    | G04      | B01       | Financial Accounting     | Walter Harrison    | Pearson          | 4th     | 2017             |            6 |                6 | F1
 BK07    | G07      | B02       | Multimedia: Making It Work | Tay Vaughan      | McGraw-Hill      | 9th     | 2014             |            5 |                5 | F2
 BK08    | G03      | B01       | Principles of Marketing  | Philip Kotler      | Pearson          | 17th    | 2020             |            9 |                9 | G1
 BK09    | G09      | B01       | Mass Communication       | Joseph Dominick    | McGraw-Hill      | 12th    | 2012             |            4 |                4 | G2
 BK10    | G02      | B02       | Understanding Business   | William Nickels    | McGraw-Hill      | 11th    | 2016             |            7 |                7 | H1
 BK11    | G05      | B02       | IT for Management        | Efraim Turban      | Wiley            | 10th    | 2018             |            5 |                5 | H2
(10 rows)
```

5. loan Table

```
CREATE TABLE loan (
   loan_id CHAR(5) PRIMARY KEY,
   user_id CHAR(5) NOT NULL,
   checkout_date DATE NOT NULL,
   due_date DATE NOT NULL,
   loan_status VARCHAR(20) NOT NULL,
   FOREIGN KEY (user_id) REFERENCES users(user_id)
);

INSERT INTO loan VALUES ('L001', 'U001', '2025-05-01', '2025-05-15', 'active');
INSERT INTO loan VALUES ('L002', 'U002', '2025-05-02', '2025-05-16', 'returned');
INSERT INTO loan VALUES ('L003', 'U003', '2025-05-03', '2025-05-17', 'overdue');
INSERT INTO loan VALUES ('L004', 'U004', '2025-05-04', '2025-05-18', 'active');
INSERT INTO loan VALUES ('L005', 'U005', '2025-05-05', '2025-05-19', 'returned');
```

```
tc3l=# select * from loan;
 loan_id | user_id |    checkout_date    |      due_date       | loan_status
---------+---------+---------------------+---------------------+-------------
 L001    | U001    | 2025-05-01 00:00:00 | 2025-05-15 00:00:00 | active
 L002    | U002    | 2025-05-02 00:00:00 | 2025-05-16 00:00:00 | returned
 L003    | U003    | 2025-05-03 00:00:00 | 2025-05-17 00:00:00 | overdue
 L004    | U004    | 2025-05-04 00:00:00 | 2025-05-18 00:00:00 | active
 L005    | U005    | 2025-05-05 00:00:00 | 2025-05-19 00:00:00 | returned
(5 rows)
```

6. fine Table

```
CREATE TABLE fine (
   fine_id CHAR(5) PRIMARY KEY,
   loan_id CHAR(5) NOT NULL,
   fine_type VARCHAR(20) NOT NULL,
   amount NUMERIC(6,2) NOT NULL,
   issue_date DATE NOT NULL,
   payment_date DATE,
   status VARCHAR(20) NOT NULL,
   FOREIGN KEY (loan_id) REFERENCES loan(loan_id)
);

INSERT INTO fine VALUES ('F001', 'L001', 'late return', 5.00, '2025-05-20', NULL,
'unpaid');
INSERT INTO fine VALUES ('F002', 'L002', 'book damage', 10.00, '2025-05-17',
'2025-05-18', 'paid');
INSERT INTO fine VALUES ('F003', 'L003', 'lost book', 25.00, '2025-05-21', NULL,
'unpaid');
INSERT INTO fine VALUES ('F004', 'L004', 'late return', 3.50, '2025-05-22', '2025-05-23',
'paid');
```

INSERT INTO fine VALUES ('F005', 'L005', 'book damage', 6.75, '2025-05-24', NULL, 'unpaid');

```
tc3l=#  select * from fine;
 fine_id | loan_id |  fine_type  | amount |     issue_date      |    payment_date     | status
---------+---------+-------------+--------+---------------------+---------------------+--------
 F001    | L001    | late return |   5.00 | 2025-05-20 00:00:00 |                     | unpaid
 F002    | L002    | book damage |  10.00 | 2025-05-17 00:00:00 | 2025-05-18 00:00:00 | paid
 F003    | L003    | lost book   |  25.00 | 2025-05-21 00:00:00 |                     | unpaid
 F004    | L004    | late return |   3.50 | 2025-05-22 00:00:00 | 2025-05-23 00:00:00 | paid
 F005    | L005    | book damage |   6.75 | 2025-05-24 00:00:00 |                     | unpaid
(5 rows)
```

7. loan_detail Table

CREATE TABLE loan_detail (
  loanDetail_ID CHAR(5) PRIMARY KEY,
  loan_id CHAR(5) NOT NULL,
  book_id CHAR(5) NOT NULL,
  returned_date DATE,
  renewed_count NUMERIC(2) NOT NULL,
  FOREIGN KEY (loan_id) REFERENCES loan(loan_id),
  FOREIGN KEY (book_id) REFERENCES book(book_id)
);

INSERT INTO loan_detail VALUES ('LD01', 'L001', 'BK01', NULL, 0);
INSERT INTO loan_detail VALUES ('LD02', 'L002', 'BK02', '2025-05-16', 1);
INSERT INTO loan_detail VALUES ('LD03', 'L003', 'BK03', NULL, 0);
INSERT INTO loan_detail VALUES ('LD04', 'L004', 'BK04', NULL, 1);
INSERT INTO loan_detail VALUES ('LD05', 'L005', 'BK05', '2025-05-19', 2);

```
tc3l=# select * from loan_detail;
 loandetail_id | loan_id | book_id |    returned_date    | renewed_count
---------------+---------+---------+---------------------+---------------
 LD01          | L001    | BK01    |                     |             0
 LD02          | L002    | BK02    | 2025-05-16 00:00:00 |             1
 LD03          | L003    | BK03    |                     |             0
 LD05          | L005    | BK05    | 2025-05-19 00:00:00 |             2
(4 rows)
```

8. damage_report Table

CREATE TABLE damage_report (
  report_id CHAR(5) PRIMARY KEY,
  loan_id CHAR(5) NOT NULL,
  damage_type VARCHAR(50) NOT NULL,
  severity VARCHAR(20) NOT NULL,
  repair_cost NUMERIC(6,2) NOT NULL,
  FOREIGN KEY (loan_id) REFERENCES loan(loan_id)
);

INSERT INTO damage_report VALUES ('R001', 'L002', 'Water Damage', 'Severe', 15.00);
INSERT INTO damage_report VALUES ('R002', 'L003', 'Torn Pages', 'Moderate', 8.50);
INSERT INTO damage_report VALUES ('R003', 'L004', 'Cover Tear', 'Minor', 4.00);
INSERT INTO damage_report VALUES ('R004', 'L005', 'Spilled Ink', 'Moderate', 7.25);
INSERT INTO damage_report VALUES ('R005', 'L001', 'Burn Marks', 'Severe', 18.00);

```
tc3l=# select * from damage_report;
 report_id | loan_id | damage_type  | severity | repair_cost
-----------+---------+--------------+----------+-------------
 R001      | L002    | Water Damage | Severe   |       15.00
 R002      | L003    | Torn Pages   | Moderate |        8.50
 R003      | L004    | Cover Tear   | Minor    |        4.00
 R004      | L005    | Spilled Ink  | Moderate |        7.25
 R005      | L001    | Burn Marks   | Severe   |       18.00
(5 rows)
```

9. reservation Table

CREATE TABLE reservation (
  reservation_id CHAR(5) PRIMARY KEY,
  user_id CHAR(5) NOT NULL,
  book_id CHAR(5) NOT NULL,
  reserve_date DATE NOT NULL,
  expiry_date DATE NOT NULL,
  book_status VARCHAR(20) NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users(user_id),
  FOREIGN KEY (book_id) REFERENCES book(book_id)
);

INSERT INTO reservation VALUES ('RS01', 'U003', 'BK03', '2025-04-29', '2025-05-03', 'reserved');
INSERT INTO reservation VALUES ('RS02', 'U002', 'BK01', '2025-04-30', '2025-05-04', 'expired');
INSERT INTO reservation VALUES ('RS03', 'U001', 'BK05', '2025-05-01', '2025-05-05', 'reserved');
INSERT INTO reservation VALUES ('RS04', 'U004', 'BK04', '2025-05-02', '2025-05-06', 'cancelled');
INSERT INTO reservation VALUES ('RS05', 'U005', 'BK02', '2025-05-03', '2025-05-07', 'reserved');

```
tc3l=# select * from reservation;
 reservation_id | user_id | book_id |    reserve_date     |     expiry_date     | book_status
----------------+---------+---------+---------------------+---------------------+-------------
 RS01           | U003    | BK03    | 2025-04-29 00:00:00 | 2025-05-03 00:00:00 | reserved
 RS02           | U002    | BK01    | 2025-04-30 00:00:00 | 2025-05-04 00:00:00 | expired
 RS03           | U001    | BK05    | 2025-05-01 00:00:00 | 2025-05-05 00:00:00 | reserved
 RS05           | U005    | BK02    | 2025-05-03 00:00:00 | 2025-05-07 00:00:00 | reserved
(4 rows)
```

Data Manipulation

i) Aggregate functions

- **SUM**

```
SELECT
   u.user_id,
   u.user_name,
   f.issue_date,
   f.amount,
   SUM(f.amount) OVER (PARTITION BY u.user_id ORDER BY f.issue_date) AS
cumulative_fines,
   ROUND(AVG(f.amount) OVER (PARTITION BY u.user_id), 2) AS avg_fine_amount
FROM users u
JOIN loan l ON u.user_id = l.user_id
JOIN fine f ON l.loan_id = f.loan_id
ORDER BY u.user_id, f.issue_date;
```

```
 user_id |  user_name  |     issue_date      | amount | cumulative_fines | avg_fine_amount
---------+-------------+---------------------+--------+------------------+-----------------
 U001    | Alice Smith | 2025-05-20 00:00:00 |   5.00 |             5.00 |            5.00
 U002    | Bob Johnson | 2025-05-17 00:00:00 |  10.00 |            10.00 |           10.00
 U003    | Charlie Lee | 2025-05-21 00:00:00 |  25.00 |            25.00 |           25.00
 U004    | Diana Wells | 2025-05-22 00:00:00 |   3.50 |             3.50 |            3.50
 U005    | Ethan Moore | 2025-05-24 00:00:00 |   6.75 |             6.75 |            6.75
(5 rows)
```

Explanation:

```
SUM(f.amount) OVER (PARTITION BY u.user_id ORDER BY f.issue_date) AS
cumulative_fines
```

- Calculates a running total of fines for each user
- PARTITION BY u.user_id: Groups calculations by user
- ORDER BY f.issue_date: Sums fines chronologically

```
ROUND(AVG(f.amount) OVER (PARTITION BY u.user_id), 2) AS avg_fine_amount
```

- Computes the average fine amount per user
- ROUND(..., 2): Ensures exactly 2 decimal places (e.g., 12.50 instead of 12.5)
- PARTITION BY u.user_id: Calculates separately for each user

```
ORDER BY u.user_id, f.issue_date
```

- Orders results by user ID first, then by fine issue date
- Makes it easy to see each user's fine history chronologically

- **COUNT overdue**

```
SELECT
   u.user_type,
   COUNT(l.loan_id) AS total_loans,
   COUNT(CASE WHEN l.loan_status = 'overdue' THEN 1 END) AS overdue_loans,
   COUNT(CASE WHEN l.loan_status = 'returned' THEN 1 END) AS returned_loans,
   COALESCE(
      ROUND(AVG(CASE WHEN l.loan_status = 'overdue' THEN DATE_PART('day',
CURRENT_DATE - l.due_date) END), 1),
      0
   ) AS avg_days_overdue
FROM users u
JOIN loan l ON u.user_id = l.user_id
GROUP BY u.user_type;
```

```
 user_type | total_loans | overdue_loans | returned_loans | avg_days_overdue
-----------+-------------+---------------+----------------+------------------
 student   |           3 |             1 |              0 |             18.0
 staff     |           2 |             0 |              2 |                0
(2 rows)
```

Explanation:

```
FROM users u
JOIN loan l ON u.user_id = l.user_id
```

- Combines user data (users table) with loan records (loan table)
- Links them via user_id to associate each loan with its borrower

```
GROUP BY u.user_type
```

- Organizes results by user type (student/staff)
- Enables comparison between different user categories

```
COUNT(l.loan_id) AS total_loans
```

- Counts all loans per user type
- Shows overall borrowing activity

```
COUNT(CASE WHEN l.loan_status = 'overdue' THEN 1 END) AS overdue_loans
```

- Conditional count of only overdue loans
- Identifies compliance issues by user type

```
COUNT(CASE WHEN l.loan_status = 'returned' THEN 1 END) AS returned_loans
```

- Counts only successfully returned items
- Measures timely returns by user category

```
COALESCE(
   ROUND(AVG(CASE WHEN l.loan_status = 'overdue'
       THEN DATE_PART('day', CURRENT_DATE - l.due_date) END), 1),
   0
) AS avg_days_overdue
```

- DATE_PART: Calculates days between due date and today
- CASE WHEN: Only includes overdue loans in calculation
- AVG: Computes average lateness
- ROUND: Formats to 1 decimal place
- COALESCE: Shows 0 instead of NULL when no overdue loans exist

- GROUP BY…HAVING

i) Nested Aggregation (Find Genres with High Reservation Rates)

```
SELECT
   g.book_name AS genre,
   COUNT(r.reservation_id) AS total_reservations,
   COUNT(DISTINCT r.user_id) AS unique_users,
   COALESCE(
      ROUND(COUNT(r.reservation_id) * 100.0 / NULLIF(COUNT(DISTINCT
b.book_id), 0), 1),
      0
   ) AS reservation_rate
```

```
FROM genre g
LEFT JOIN book b ON g.genre_ID = b.genre_ID
LEFT JOIN reservation r ON b.book_id = r.book_id
GROUP BY g.book_name
HAVING COUNT(r.reservation_id) > 0
ORDER BY reservation_rate DESC;
```

```
     genre       | total_reservations | unique_users | reservation_rate
-----------------+--------------------+--------------+------------------
 Computer Science |                  1 |            1 |            100.0
 Engineering      |                  1 |            1 |            100.0
 Finance          |                  1 |            1 |            100.0
 Law              |                  1 |            1 |            100.0
(4 rows)
```

Explanation:

The primary goal of this query is to:

- Count total reservations per genre
- Measure unique users reserving books in each genre
- Calculate a normalized "reservation rate" (reservations per available book)
- Rank genres by popularity (highest reservation rate first)

This helps library administrators understand demand patterns and optimize book acquisitions.

There are three tables involved which are;

Table	Alias	Description
genre	g	Contains book genre names such as Law, Computer Science and other university books.
book	b	Stores book details, linked to a genre via genre_ID
reservation	r	Records book reservations by users

Join Flow:
genre (g) → book (b)

- Joins on g.genre_ID = b.genre_ID
- Ensures every genre is matched with its books (if any).

book (b) → reservation (r)

- Joins on b.book_id = r.book_id
- Links books to their reservations

By using LEFT JOIN, it ensures all genres appear in results, even if they have no books or reservations and missing data such as no books or reservations will appear as NULL but are handled by COUNT and COALESCE.

```
COUNT(r.reservation_id) AS total_reservations
```

- Counts all reservations for books in each genre.
- Only counts non-NULL reservation_id values (due to COUNT behavior).

```
COUNT(DISTINCT r.user_id) AS unique_users
```

- Counts unique users who reserved books in each genre.
- DISTINCT ensures the same user reserving multiple books isn't double-counted.

reservation_rate

As for reservation_rate calculation, the formula used was, (Total Reservations × 100) ÷ (Number of Books in Genre). It is multiplied by 100 so that it could be converted to a percentage for readability.

```
NULLIF(COUNT(DISTINCT b.book_id), 0)
```

- Prevents division by zero if a genre has no books.
- If COUNT(DISTINCT b.book_id) = 0, NULLIF returns NULL, making the whole expression NULL.

- COALESCE(..., 0) → Replaces NULL with 0 (if no books exist in a genre).
- ROUND(..., 1) → Rounds the result to 1 decimal place for cleaner reporting.

Filtering and Sorting

```
GROUP BY g.book_name
```

- Aggregates results by genre name since it is by per-genre statistics

```
HAVING COUNT(r.reservation_id) > 0
```

- Filters out genres with zero reservations (only shows genres with activity).
- Alternative: Remove this to see all genres (including those with no reservations).

```
ORDER BY reservation_rate DESC
```

- Ranks genres from highest to lowest reservation rate.
- Helps quickly identify the most in-demand genres.

ii) Filtering Aggregated Results (Users with Multiple Overdue Loans)

```
WITH loan_stats AS (
    SELECT
        u.user_id,
        u.user_name,
        u.user_type,
        u.max_borrow_limit,
        COUNT(l.loan_id) AS total_loans,
        COUNT(CASE WHEN l.loan_status = 'overdue' THEN 1 END) AS overdue_loans,
        COUNT(CASE WHEN l.loan_status = 'returned' THEN 1 END) AS returned_loans,
        SUM(CASE WHEN l.loan_status = 'overdue' THEN DATE_PART('day',
CURRENT_DATE - l.due_date) ELSE 0 END) AS total_overdue_days,
        SUM(f.amount) AS total_fines,
        MAX(CASE WHEN l.loan_status = 'overdue' THEN DATE_PART('day',
CURRENT_DATE - l.due_date) ELSE 0 END) AS max_days_overdue
    FROM users u
    JOIN loan l ON u.user_id = l.user_id
    LEFT JOIN fine f ON l.loan_id = f.loan_id
    GROUP BY u.user_id, u.user_name, u.user_type, u.max_borrow_limit
)
SELECT
    user_id,
    user_name,
    user_type,
    total_loans,
    overdue_loans,
    returned_loans,
    total_overdue_days,
    total_fines,
    max_days_overdue,
    ROUND(total_overdue_days * 1.0 / NULLIF(overdue_loans, 0), 1) AS
avg_days_overdue,
    ROUND(overdue_loans * 100.0 / NULLIF(total_loans, 0), 1) AS overdue_percentage,
    CASE
        WHEN overdue_loans > 0 THEN
            RANK() OVER (ORDER BY total_overdue_days DESC)
        ELSE NULL
    END AS severity_rank
```

```
FROM loan_stats
WHERE overdue_loans > 0
  AND total_fines > 0
ORDER BY
    severity_rank NULLS LAST,
    total_fines DESC;
```

```
tc3l-#     total_fines DESC;
 user_id |  user_name  | user_type | total_loans | overdue_loans | returned_loans | total_overdue_days |
---------+-------------+-----------+-------------+---------------+----------------+--------------------+
 U003    | Charlie Lee | student   |           1 |             1 |              0 |                 18 |
(1 row)
```

```
 | total_fines | max_days_overdue | avg_days_overdue | overdue_percentage | severity_rank
-+-------------+------------------+------------------+--------------------+---------------
 |       25.00 |               18 |             18.0 |              100.0 |             1
```

Explanation:
This SQL query performs an in-depth analysis of user loan behavior, focusing on overdue patterns and associated fines. It utilizes a Common Table Expression (CTE) to compute key loan metrics before applying additional calculations and ranking. The results help identify users with the most severe overdue issues, enabling targeted follow-up actions**.**

Main query:

Metric	Description
avg_days_overdue	Calculates the average days overdue per loan by using this formula (total_overdue_days / overdue_loans)
overdue_percentage	Displays the percentage of loans that are overdue by calculating with (overdue_loans / total_loans × 100)
severity_rank	Ranks users by how severe the book damage
Filters	Only includes users with: - At least one overdue loan (overdue_loans > 0) - Outstanding fines (total_fines > 0)
Sorting	It is order by the highest which is severity_rank and then descending which is

	total_fines

Line-by-line query explanation

```
WITH loan_stats AS (
   SELECT
      u.user_id,
      u.user_name,
      u.user_type,
      u.max_borrow_limit,
```

- Creates a temporary result set named loan_stats that exists only during query execution
- Selects user attributes: unique ID, name, classification type, and maximum borrowing allowance

```
COUNT(l.loan_id) AS total_loans,
```

- Counts all loan transactions per user using loan_id as the counting key
- Includes all loans regardless of status (active, overdue, or returned)

```
COUNT(CASE WHEN l.loan_status = 'overdue' THEN 1 END) AS overdue_loans,
```

- Conditional count that only increments when loan_status = 'overdue'
- The CASE expression acts as a filter within the aggregation
- Returns count of currently overdue items per user

```
COUNT(CASE WHEN l.loan_status = 'returned' THEN 1 END) AS returned_loans,
```

- Similar conditional count for successfully returned items
- Checks for status 'returned' (verify spelling matches with the actual data)

```
SUM(CASE WHEN l.loan_status = 'overdue' THEN DATE_PART('day',
CURRENT_DATE - l.due_date) ELSE 0 END) AS total_overdue_days,
```

- For overdue items, calculates days overdue as: current date - due date
- DATE_PART('day', ...) extracts just the day component from the interval
- SUM aggregates the total overdue days across all loans
- Non-overdue loans contribute 0 to the sum

```
SUM(f.amount) AS total_fines,
```

- Sums all fine amounts associated with the user's loans
- LEFT JOIN ensures users without fines are included (sum will be NULL, treated as 0)

```
 MAX(CASE WHEN l.loan_status = 'overdue' THEN DATE_PART('day',
CURRENT_DATE - l.due_date) ELSE 0 END) AS max_days_overdue
```

- Finds the single most overdue item for each user
- MAX returns the largest value from the conditional date calculation
- Shows the worst-case delinquency for priority handling

```
  FROM users u
   JOIN loan l ON u.user_id = l.user_id
   LEFT JOIN fine f ON l.loan_id = f.loan_id
   GROUP BY u.user_id, u.user_name, u.user_type, u.max_borrow_limit
)
```

- INNER JOIN between users and loans ensures we only consider users with loan activity
- LEFT JOIN to fines preserves all loans even if no fine exists
- GROUP BY consolidates results at user level while preserving selected attributes

```
SELECT
   user_id,
   user_name,
   user_type,
   total_loans,
   overdue_loans,
   returned_loans,
   total_overdue_days,
   total_fines,
   max_days_overdue,
```

- Retrieves all basic metrics from the CTE
- Includes both raw counts and calculated sums from the first stage

```
ROUND(total_overdue_days * 1.0 / NULLIF(overdue_loans, 0), 1) AS avg_days_overdue,
```

- Calculates average days overdue per delinquent loan

- NULLIF protects against division by zero if no overdue loans exist
- ROUND(..., 1) formats to 1 decimal place for readability
- Multiplying by 1.0 ensures floating-point division

```
ROUND(overdue_loans * 100.0 / NULLIF(total_loans, 0), 1) AS overdue_percentage,
```

- Computes what percentage of user's total loans are overdue
- Scales to percentage by multiplying by 100.0
- Again uses NULLIF to avoid division by zero errors

```
  CASE
     WHEN overdue_loans > 0 THEN
        RANK() OVER (ORDER BY total_overdue_days DESC)
     ELSE NULL
  END AS severity_rank
```

- Conditional ranking only applied to users with overdue items
- RANK() assigns positions based on total overdue days (highest = 1)
- Users with no overdues receive NULL instead of a rank

```
FROM loan_stats
WHERE overdue_loans > 0
  AND total_fines > 0
```

- Filters final output to only show delinquent users
- First condition: Must have at least one overdue loan
- Second condition: Must have outstanding fines (positive amount)

```
ORDER BY
   severity_rank NULLS LAST,
   total_fines DESC;
```

- Primary sort: Users ranked by severity (most severe first)
- NULLS LAST ensures properly ranked users appear before any NULLs
- Secondary sort: When ranks are equal, higher fines appear first
- DESC (descending) orders larger values before smaller ones

iii) Trigger

BEFORE

```
CREATE OR REPLACE FUNCTION validate_loan_conditions()
RETURNS TRIGGER AS $$
DECLARE
   v_user_status TEXT;
   v_current_loans INT;
   v_max_limit INT;
BEGIN

   SELECT account_status INTO v_user_status
   FROM users WHERE user_id = NEW.user_id;

   IF v_user_status = 'suspended' THEN
      RAISE EXCEPTION 'User % is suspended and cannot borrow books', NEW.user_id;
   END IF;


   SELECT
      COUNT(*),
      u.max_borrow_limit
   INTO v_current_loans, v_max_limit
   FROM loan l
   JOIN users u ON l.user_id = u.user_id
   WHERE l.user_id = NEW.user_id AND l.loan_status = 'active'
   GROUP BY u.max_borrow_limit;

   IF v_current_loans >= v_max_limit THEN
      RAISE EXCEPTION 'User % has reached the maximum borrow limit (%)',
NEW.user_id, v_max_limit;
   END IF;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_validate_loan
BEFORE INSERT ON loan
FOR EACH ROW
EXECUTE PROCEDURE validate_loan_conditions();
```

AFTER

```
CREATE OR REPLACE FUNCTION update_book_availability()
RETURNS TRIGGER AS $$
BEGIN

   UPDATE book
   SET available_copies = available_copies - 1
```

```
    WHERE book_id = NEW.book_id;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER after_loan_detail_insert
AFTER INSERT ON loan_detail
FOR EACH ROW
EXECUTE PROCEDURE update_book_availability();
```

```
tc3l=# select * from loan;
 loan_id | user_id |    checkout_date    |      due_date       | loan_status
---------+---------+---------------------+---------------------+-------------
 L001    | U001    | 2025-05-01 00:00:00 | 2025-05-15 00:00:00 | active
 L002    | U002    | 2025-05-02 00:00:00 | 2025-05-16 00:00:00 | returned
 L003    | U003    | 2025-05-03 00:00:00 | 2025-05-17 00:00:00 | overdue
 L004    | U004    | 2025-05-04 00:00:00 | 2025-05-18 00:00:00 | active
 L005    | U005    | 2025-05-05 00:00:00 | 2025-05-19 00:00:00 | returned
(5 rows)

tc3l=# UPDATE users SET account_status = 'suspended' WHERE user_id = 'U005';
UPDATE 1
tc3l=# INSERT INTO loan VALUES ('L006', 'U005', '2025-05-06', '2025-05-20', 'active');
ERROR:  User U005  is suspended and cannot borrow books
```

For context

UPDATE users SET account_status = ‘suspended’ WHERE user_id = ‘U005’;
INSERT INTO loan VALUES (‘L006’, ‘U005’, ‘2025-05-06’, ‘2023-05-20’, ‘active’);
Shows error when trying insert values when account is suspended

```
tc3l=# INSERT INTO loan VALUES ('L008', 'U006', '2025-06-12', '2025-06-26', 'active');
INSERT 0 1
tc3l=# INSERT INTO loan VALUES ('L009', 'U006', '2025-06-13', '2025-06-27', 'active');
INSERT 0 1
tc3l=# select * from loan;
 loan_id | user_id |    checkout_date    |      due_date       | loan_status
---------+---------+---------------------+---------------------+-------------
 L001    | U001    | 2025-05-01 00:00:00 | 2025-05-15 00:00:00 | active
 L002    | U002    | 2025-05-02 00:00:00 | 2025-05-16 00:00:00 | returned
 L003    | U003    | 2025-05-03 00:00:00 | 2025-05-17 00:00:00 | overdue
 L004    | U004    | 2025-05-04 00:00:00 | 2025-05-18 00:00:00 | active
 L005    | U005    | 2025-05-05 00:00:00 | 2025-05-19 00:00:00 | returned
 L007    | U006    | 2025-06-11 00:00:00 | 2025-06-25 00:00:00 | active
 L008    | U006    | 2025-06-12 00:00:00 | 2025-06-26 00:00:00 | active
 L009    | U006    | 2025-06-13 00:00:00 | 2025-06-27 00:00:00 | active
(8 rows)

tc3l=# INSERT INTO loan VALUES ('L010', 'U006', '2025-06-14', '2025-06-28', 'active');
ERROR:  Limit reached (max 3)
```

For context

INSERT INTO users VALUES
('U006', 'Lisa Ray', 'lisa@example.com', '0112233445', 'student', 3, 'active');

INSERT INTO loan VALUES ('L007', 'U006', '2025-06-11', '2025-06-25', 'active');
INSERT INTO loan VALUES ('L008', 'U006', '2025-06-12', '2025-06-26', 'active');
INSERT INTO loan VALUES ('L009', 'U006', '2025-06-13', '2025-06-27', 'active');

Shows error when user tries to borrow more than 3 books box as max_borrow_limit is set to 3

```
tc3l=# SELECT book_id, title, available_copies
tc3l-# FROM book
tc3l-# WHERE book_id = 'BK01';
 book_id |      title       | available_copies
---------+------------------+------------------
 BK01    | The Time Machine |                8
(1 row)

tc3l=#
```

Explanation:

BEFORE

```
CREATE OR REPLACE FUNCTION validate_loan_conditions()
RETURNS TRIGGER AS $$
```

- Creates or replaces a function named validate_loan_conditions
- Specifies that this function returns a trigger object (required for trigger functions)
- $$ marks the beginning of the function body

```
DECLARE
   v_user_status TEXT;
   v_current_loans INT;
   v_max_limit INT;
```

- v_user_status to store the user's account status (text)
- v_current_loans to count current active loans (integer)
- v_max_limit to store the user's maximum borrow limit (integer)

```
  SELECT account_status INTO v_user_status
   FROM users WHERE user_id = NEW.user_id;

   IF v_user_status = 'suspended' THEN
      RAISE EXCEPTION 'User % is suspended and cannot borrow books', NEW.user_id;
   END IF;
```

- Retrieves the account status of the user attempting to borrow a book (NEW.user_id)
- Stores the status in v_user_status
- If the status is 'suspended', raises an exception preventing the loan

- NEW refers to the new row being inserted into the loan table

```
  SELECT
    COUNT(*),
    u.max_borrow_limit
  INTO v_current_loans, v_max_limit
  FROM loan l
  JOIN users u ON l.user_id = u.user_id
  WHERE l.user_id = NEW.user_id AND l.loan_status = 'active'
  GROUP BY u.max_borrow_limit;
```

- Counts all active loans for the user and gets their maximum borrow limit

- Joins the loan table with users table on user_id
- Filters for the current user and only active loans
- Stores the count in v_current_loans and the limit in v_max_limit

```
  IF v_current_loans >= v_max_limit THEN
      RAISE EXCEPTION 'User % has reached the maximum borrow limit (%)',
NEW.user_id, v_max_limit;
  END IF;
```

- Checks if current loans meet or exceed the user's maximum limit
- If true, raises an exception with a descriptive message showing user_id and limit

```
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

- Returns the NEW row (allowing the insert to proceed if no exceptions were raised)
- Marks the end of the function body with END;
- $$ closes the function body
- Specifies the language as PL/pgSQL (PostgreSQL's procedural language)

```
CREATE TRIGGER trg_validate_loan
BEFORE INSERT ON loan
FOR EACH ROW
EXECUTE PROCEDURE validate_loan_conditions();
```

- Creates a trigger named trg_validate_loan

- Specifies it fires BEFORE any INSERT operation on the loan table
- Configures it to execute FOR EACH ROW (row-level trigger)
- Associates it with the validate_loan_conditions() function

AFTER

```
CREATE OR REPLACE FUNCTION update_book_availability()
```

- CREATE OR REPLACE FUNCTION: Creates a new function or replaces an existing one with the same name.
- update_book_availability(): The name of the function we're creating.

```
RETURNS TRIGGER AS $$
```

- RETURNS TRIGGER: Specifies that this function returns a trigger type (required for trigger functions).
- AS $$: Begins the function body (the $$ are delimiters for the function code).

```
-- Decrease available copies when a book is loaned
UPDATE book
SET available_copies = available_copies - 1
WHERE book_id = NEW.book_id;
```

- UPDATE book: Specifies we're updating the book table.
- SET available_copies = available_copies - 1: Decrements the available_copies by 1.
- WHERE book_id = NEW.book_id: Only updates the book whose ID matches the newly inserted loan_detail record's book_id.
- NEW: A special trigger variable representing the new row being inserted.

```
RETURN NEW;
```

- Returns the modified row (required for AFTER triggers, though the value is actually ignored).

```
END;
$$ LANGUAGE plpgsql;
```

- END;: Marks the end of the function body.
- LANGUAGE plpgsql: Specifies that the function is written in PL/pgSQL.

```
CREATE TRIGGER after_loan_detail_insert
```

- CREATE TRIGGER: Creates a new trigger.
- after_loan_detail_insert: The name we're giving to this trigger.

```
AFTER INSERT ON loan_detail
```

- AFTER INSERT: Specifies the trigger fires after an insert operation completes.
- ON loan_detail: The table this trigger is attached to.

```
FOR EACH ROW
```

- Indicates this is a row-level trigger (executes once for each affected row).

```
EXECUTE PROCEDURE update_book_availability();
```

- EXECUTE PROCEDURE: OpenGauss syntax to specify the function to execute (note: PostgreSQL would use EXECUTE FUNCTION).
- update_book_availability(): Calls the function we defined earlier.

iv) Stored procedure

```
CREATE OR REPLACE FUNCTION checkout_book(
    p_user_id CHAR(5),
    p_book_id CHAR(5),
    p_checkout_date DATE DEFAULT CURRENT_DATE
) RETURNS VARCHAR(100) AS $$
DECLARE
    v_loan_id CHAR(5);
    v_available_copies INT;
    v_user_status VARCHAR(10);
    v_current_loans INT;
    v_max_limit INT;
    v_due_date DATE;
BEGIN

    SELECT account_status, max_borrow_limit INTO v_user_status, v_max_limit
    FROM users WHERE user_id = p_user_id;

    IF v_user_status != 'active' THEN
        RETURN 'Error: User account is not active';
    END IF;
```

```
    SELECT COUNT(*) INTO v_current_loans
    FROM loan WHERE user_id = p_user_id AND loan_status = 'active';

    IF v_current_loans >= v_max_limit THEN
        RETURN 'Error: User has reached maximum borrowing limit';
    END IF;


    SELECT available_copies INTO v_available_copies
    FROM book WHERE book_id = p_book_id;

    IF v_available_copies <= 0 THEN
        RETURN 'Error: Book not available';
    END IF;


    SELECT 'L' || LPAD((COALESCE(MAX(CAST(SUBSTRING(loan_id, 2) AS INT)), 0)
+ 1)::TEXT, 3, '0')
    INTO v_loan_id FROM loan;


    v_due_date := p_checkout_date + INTERVAL '14 days';


    INSERT INTO loan (loan_id, user_id, checkout_date, due_date, loan_status)
    VALUES (v_loan_id, p_user_id, p_checkout_date, v_due_date, 'active');


    INSERT INTO loan_detail (loanDetail_ID, loan_id, book_id, returned_date,
renewed_count)
    VALUES ('LD' || SUBSTRING(v_loan_id, 2), v_loan_id, p_book_id, NULL, 0);


    UPDATE book SET available_copies = available_copies - 1
    WHERE book_id = p_book_id;

    RETURN 'Success: Book checked out. Loan ID: ' || v_loan_id || ', Due: ' || v_due_date;
END;
$$ LANGUAGE plpgsql;
```

Explanation:

Before:

```
CREATE OR REPLACE FUNCTION checkout_book(
    p_user_id CHAR(5),
    p_book_id CHAR(5),
    p_checkout_date DATE DEFAULT CURRENT_DATE
)
RETURNS VARCHAR(100) AS $$
```

-Creates or replaces a function named checkout_book

-Takes input parameters: p_user_id, p_book_id, and optional p_checkout_date

-Returns a VARCHAR(100) message string

-$$ marks the start of the function body

Declare:

```
 DECLARE
   v_loan_id CHAR(5);
   v_available_copies INT;
   v_user_status VARCHAR(10);
   v_current_loans INT;
   v_max_limit INT;
   v_due_date DATE;
```

-v_loan_id: new loan ID
-v_available_copies: book stock count
-v_user_status: user's account status
-v_current_loans: count of current active loans
-v_max_limit: maximum borrow limit for the user
-v_due_date: calculated due date

Check user status:

```
SELECT account_status, max_borrow_limit INTO v_user_status, v_max_limit
FROM users WHERE user_id = p_user_id;
```

-Retrieves user’s account status and borrow limit

-Stores them into local variables v_user_status and v_max_limit

```
IF v_user_status != 'active' THEN
    RETURN 'Error: User account is not active';
END IF;
```

-If the user is not active, the function returns an error message
-Prevents checkout for suspended or inactive users

Check current loan count:

```
SELECT COUNT(*) INTO v_current_loans
FROM loan WHERE user_id = p_user_id AND loan_status = 'active';
```

-Counts how many active loans the user currently has

-Stores the result in v_current_loans

```
IF v_current_loans >= v_max_limit THEN
    RETURN 'Error: User has reached maximum borrowing limit';
END IF;
```

-Compares current loans to the max borrow limit

-Returns an error if the user has already reached their limit

Check book availability:

```
SELECT available_copies INTO v_available_copies
FROM book WHERE book_id = p_book_id;
```

-Checks how many copies of the book are available
-Stores the value in v_available_copies

```
IF v_available_copies <= 0 THEN
    RETURN 'Error: Book not available';
```

```
END IF;
```

-If no copies are available, return an error

-Prevents over-lending a book

Generate loan ID:

```
SELECT 'L' || LPAD((COALESCE(MAX(CAST(SUBSTRING(loan_id, 2) AS INT)), 0) +
1)::TEXT, 3, '0')
INTO v_loan_id FROM loan;
```

Generates a new loan ID by:

-Taking the max existing loan ID (numeric part only)
-Adding 1, and padding with zeros to 3 digits

Set due date:

```
v_due_date := p_checkout_date + INTERVAL '14 days';
```

-Sets due date to 14 days after checkout

-Uses p_checkout_date (defaults to CURRENT_DATE)

Insert into loan table:

```
INSERT INTO loan (loan_id, user_id, checkout_date, due_date, loan_status)
VALUES (v_loan_id, p_user_id, p_checkout_date, v_due_date, 'active');
```

-Inserts a new record into the loan table
-Marks the loan as active

Insert into loan detail table:

```
INSERT INTO loan_detail (loanDetail_ID, loan_id, book_id, returned_date,
renewed_count)
VALUES ('LD' || SUBSTRING(v_loan_id, 2), v_loan_id, p_book_id, NULL, 0);
```

-Inserts a record into loan_detail

-Uses part of loan_id to create a corresponding loan detail ID

-Initializes returned_date as NULL and renewed_count as 0

Update book availability:

```
UPDATE book SET available_copies = available_copies - 1
WHERE book_id = p_book_id;
```

-Reduces the available copies of the borrowed book by 1

Return success message:

```
RETURN 'Success: Book checked out. Loan ID: ' || v_loan_id || ', Due: ' || v_due_date;
```

-Returns a message confirming successful checkout

-Includes the new loan ID and due date in the message

```
END;
$$ LANGUAGE plpgsql;
```

-Ends the function body

v) View

Example: A view showing user names, their loan IDs, and corresponding fine amounts (if any).

Explanation: To make it easier to see which users have fines, we create a view combining user names, loan IDs, fine amounts, and their status.

```
CREATE OR REPLACE VIEW user_loan_fine_view AS SELECT u.user_name, l.loan_id, f.amount AS fine_amount, f.status AS fine_status FROM users u JOIN loan l ON u.user_id = l.user_id LEFT JOIN fine f ON l.loan_id = f.loan_id;
```

```
tc3l=# select * from user_loan_fine_view;
  user_name   | loan_id | fine_amount | fine_status
--------------+---------+-------------+-------------
 Alice Smith  | L001    |        5.00 | unpaid
 Bob Johnson  | L002    |       10.00 | paid
 Charlie Lee  | L003    |       25.00 | unpaid
 Diana Wells  | L004    |        3.50 | paid
 Ethan Moore  | L005    |        6.75 | unpaid
```

```
CREATE OR REPLACE VIEW user_loan_fine_view AS
SELECT
  u.user_name,
  l.loan_id,
  f.amount AS fine_amount,
  f.status AS fine_status
FROM users u
JOIN loan l ON u.user_id = l.user_id
LEFT JOIN fine f ON l.loan_id = f.loan_id;
```

-Creates or replaces a view named user_loan_fine_view

-The view presents a joined display of user names, loan IDs, and any associated fine information

-SELECT u.user_name, l.loan_id, f.amount, f.status specifies the columns to display:

- user_name: from users table

- loan_id: from loan table
- fine_amount: from fine.amount (aliased)
- fine_status: from fine.status (aliased)

-FROM users u starts the query from the users table, given alias u

-JOIN loan l ON u.user_id = l.user_id: performs an inner join to get all loans tied to each user

-LEFT JOIN fine f ON l.loan_id = f.loan_id: includes all loans, even those without a fine, using a left join

-Ensures that loans without fines will show NULL in fine_amount and fine_status

v) Subqueries/nested queries

Example 1: Users who have borrowed more books than the average number of books borrowed per user.

```
SELECT u.user_id, u.user_name
FROM users u
WHERE (
   SELECT COUNT(*)
   FROM loan l
   JOIN loan_detail ld ON l.loan_id = ld.loan_id
   WHERE l.user_id = u.user_id
) > (
   SELECT AVG(book_count)
   FROM (
      SELECT COUNT(*) AS book_count
      FROM loan l
      JOIN loan_detail ld ON l.loan_id = ld.loan_id
      GROUP BY l.user_id
   ) AS user_book_counts
);

) > u.max_borrow_limit;
```

Explanation:

```
SELECT u.user_id, u.user_name
```

-Selects the user_id and user_name from the users table

-These are the details we want to retrieve for users who meet a certain condition

```
FROM users u
```

-Specifies the source table users with alias u

-This alias is used to refer to the users table in other parts of the query

```
WHERE (
   SELECT COUNT(*)
   FROM loan l
   JOIN loan_detail ld ON l.loan_id = ld.loan_id
   WHERE l.user_id = u.user_id
)
```

-Subquery to count how many books the user has borrowed in total

-Joins loan and loan_detail to access book-level details of each loan

-Filters loans where loan.user_id matches the current user u.user_id in the outer query

```
> (
```

```
   SELECT AVG(book_count)
   FROM (
      SELECT COUNT(*) AS book_count
      FROM loan l
      JOIN loan_detail ld ON l.loan_id = ld.loan_id
      GROUP BY l.user_id
   ) AS user_book_counts
);
```

-Compares the user's total book count to the average number of books borrowed per user

-Inner subquery groups by each user and counts their borrowed books

-Outer subquery calculates the average of these counts

-Final result: selects users who have borrowed **more books than the average**

Example 2: Total fines paid and unpaid by status for each user.

```
SELECT u.user_id, u.user_name,
      SUM(CASE WHEN f.status = 'paid' THEN f.amount ELSE 0 END) AS total_paid,
      SUM(CASE WHEN f.status = 'unpaid' THEN f.amount ELSE 0 END) AS
total_unpaid
FROM users u
JOIN loan l ON u.user_id = l.user_id
JOIN fine f ON l.loan_id = f.loan_id
GROUP BY u.user_id, u.user_name;
```

Explanation:

```
SELECT u.user_id, u.user_name,
```

-Selects the user_id and user_name from the users table

-These are the basic user details to be displayed in the result

```
    SUM(CASE WHEN f.status = 'paid' THEN f.amount ELSE 0 END) AS total_paid,
```

-Sums up all fine amounts where the fine status is `'paid'`

-Uses a `CASE` statement to include the amount only when status is `'paid'`

-Returns the total as `total_paid` for each user

```
    SUM(CASE WHEN f.status = 'unpaid' THEN f.amount ELSE 0 END) AS
total_unpaid
```

-Sums up all fine amounts where the fine status is 'unpaid'

-Uses a CASE statement to include the amount only when status is 'unpaid'

-Returns the total as total_unpaid for each use

```
FROM users u
```

-Specifies the `users` table as the source

-Gives it the alias `u` to reference later in the query

```
JOIN loan l ON u.user_id = l.user_id
```

-Joins the loan table with the users table on the user_id column

-Ensures each user is connected to their loans

```
JOIN fine f ON l.loan_id = f.loan_id
```

-Joins the fine table with the loan table on the loan_id column

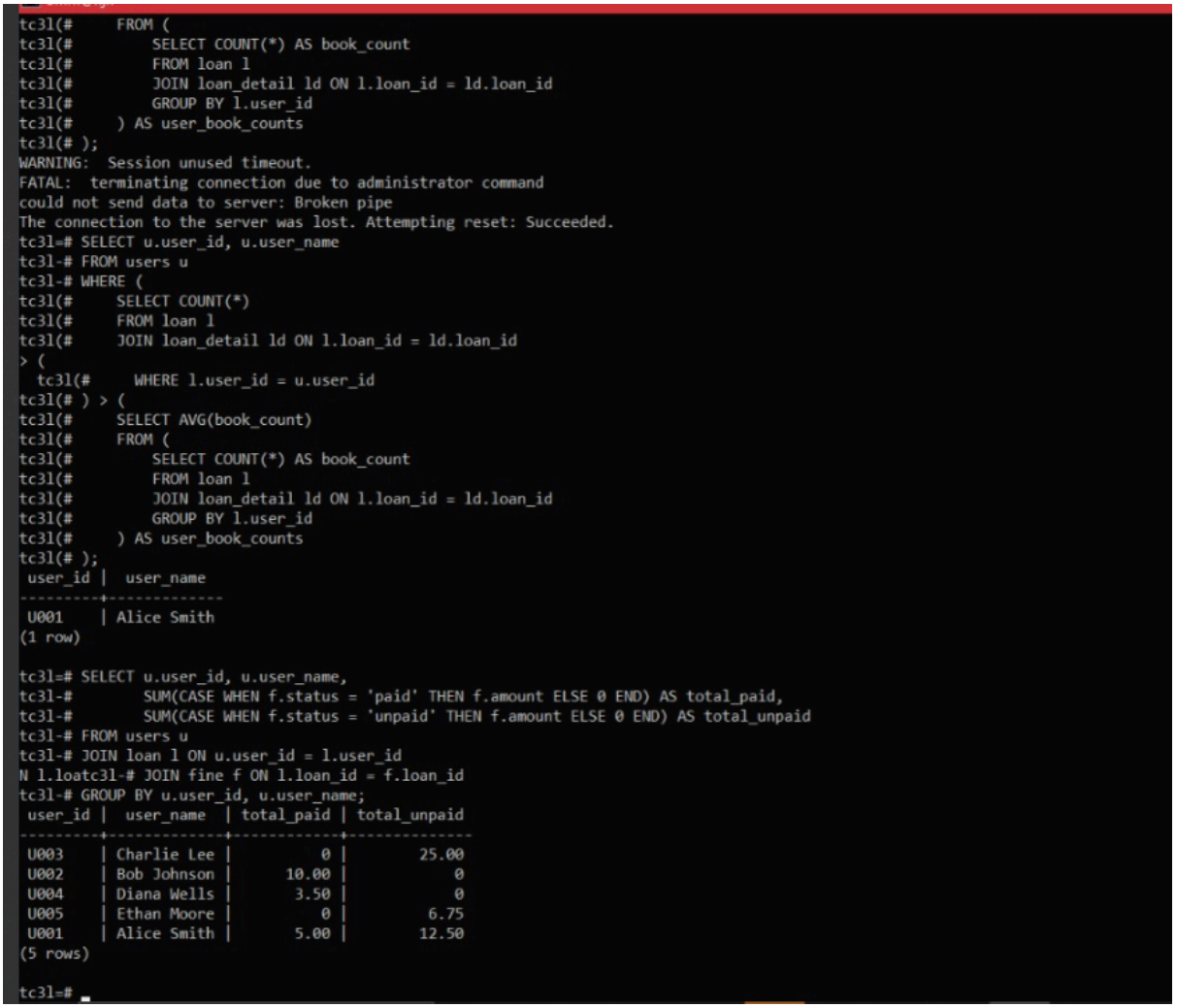
-Ensures each loan is connected to its associated fines

```
GROUP BY u.user_id, u.user_name;
```

-Groups the results by user_id and user_name

-Required when using aggregate functions like SUM()

-Ensures each row in the output represents a single user with their total fines



Example 3: Correlated Subquery to Find Users with All Books Overdue

```
SELECT u.user_id, u.user_name, u.email
FROM users u
WHERE u.account_status = 'active'
AND NOT EXISTS (
   -- Find any active loan that's NOT overdue for this user
   SELECT 1
   FROM loan l
   WHERE l.user_id = u.user_id
   AND l.loan_status = 'active'
   AND CURRENT_DATE <= l.due_date
)
AND EXISTS (
   -- Ensure the user has at least one active loan
   SELECT 1
   FROM loan l
   WHERE l.user_id = u.user_id
   AND l.loan_status = 'active'
)
```

```
ORDER BY u.user_name;
```

Explanation:

```
SELECT u.user_id, u.user_name, u.email
```

-Selects user details: user_id, user_name, and email

-These are the output fields that will be shown in the final result

```
FROM users u
```

-Specifies the users table as the source of the data

-Uses alias u to refer to the users table throughout the query

```
WHERE u.account_status = 'active'
```

-Filters the users to include only those with an account status of 'active'

-Ensures that only currently active users are considered

```
AND NOT EXISTS (
```

-Begins a subquery that uses NOT EXISTS to exclude certain users

-The condition inside will define which users should be excluded

```
    SELECT 1
    FROM loan l
    WHERE l.user_id = u.user_id
    AND l.loan_status = 'active'
    AND CURRENT_DATE <= l.due_date
)
```

-Checks for any non-overdue active loans for the user

-If such a loan exists, the user will not be included in the result

-CURRENT_DATE <= l.due_date means the loan is not yet overdue

-So this part ensures all active loans for the user are overdue

```
AND EXISTS (
```

-Begins another subquery using EXISTS

-This subquery ensures the user has at least one active loan

```
    SELECT 1
    FROM loan l
    WHERE l.user_id = u.user_id
    AND l.loan_status = 'active'
)
```

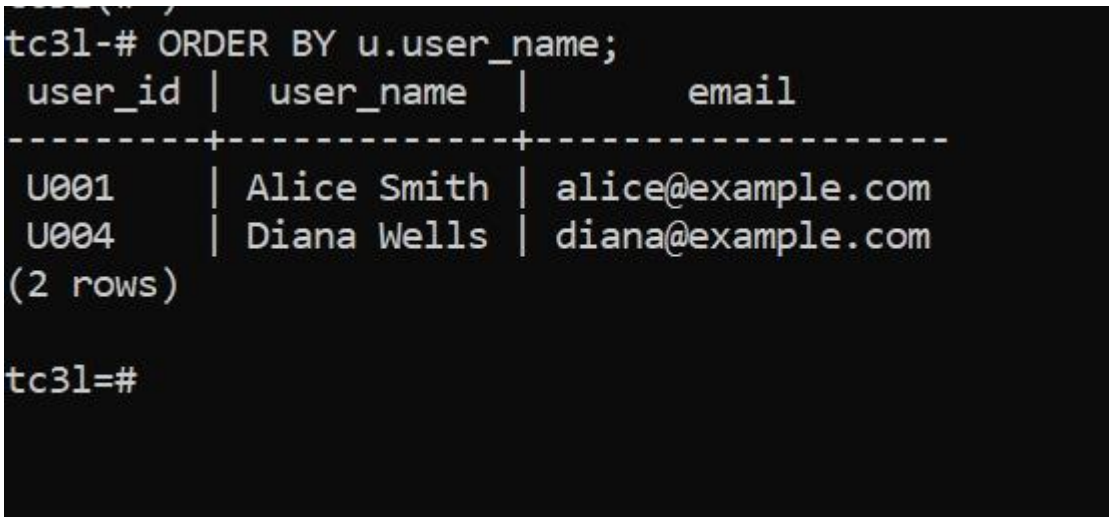
```

```

-Confirms that the user has at least one active loan
-Ensures we only include users who are actually borrowing something

```
ORDER BY u.user_name;
```

Sorts the final result alphabetically by user_name



Example 4: Nested Subquery with Window Function to Find Popular Genres

```
WITH genre_loan_stats AS (
   SELECT
      g.genre_ID,
      g.book_name AS genre_name,
      COUNT(ld.loanDetail_ID) AS total_loans,
      COUNT(DISTINCT ld.book_id) AS unique_books_loaned,
      COUNT(DISTINCT l.user_id) AS unique_borrowers,
      ROUND(COUNT(ld.loanDetail_ID) * 100.0 / SUM(COUNT(ld.loanDetail_ID))
OVER(), 2) AS percentage_of_total
   FROM genre g
   JOIN book b ON g.genre_ID = b.genre_ID
```

```
    JOIN loan_detail ld ON b.book_id = ld.book_id
    JOIN loan l ON ld.loan_id = l.loan_id
    GROUP BY g.genre_ID, g.book_name
)
SELECT
    genre_ID,
    genre_name,
    total_loans,
    unique_books_loaned,
    unique_borrowers,
    percentage_of_total || '%' AS percentage_of_total_loans,
    DENSE_RANK() OVER (ORDER BY total_loans DESC) AS popularity_rank
FROM genre_loan_stats
ORDER BY total_loans DESC
LIMIT 3;
```

Explanation:

Before

```
WITH genre_loan_stats AS (
```

-Starts a **Common Table Expression (CTE)** named genre_loan_stats

-This is a temporary result set that can be referenced later in the query

```
    SELECT
        g.genre_ID,
        g.book_name AS genre_name,
```

-Selects the genre ID and genre name from the genre table

-book_name is assumed to represent the genre label (possibly misnamed column)

```
        COUNT(ld.loanDetail_ID) AS total_loans,
```

```

```

-Counts how many times books in this genre were loaned

-loanDetail_ID ensures each individual loan detail is counted

```
    COUNT(DISTINCT ld.book_id) AS unique_books_loaned,
```

-Counts the number of **distinct books** that were loaned in this genre

-Helps identify how many different books from this genre were borrowed

```
    COUNT(DISTINCT l.user_id) AS unique_borrowers,
```

-Counts how many **distinct users** borrowed books from this genre

-Measures the reach or popularity of the genre across users

```
    ROUND(COUNT(ld.loanDetail_ID) * 100.0 / SUM(COUNT(ld.loanDetail_ID)) OVER(), 2) AS percentage_of_total
```

-Calculates the percentage of total loans this genre contributes

-Uses a WINDOW FUNCTION (SUM(...) OVER()) to compute total across all genres

-Multiplies by 100 to convert to percentage, rounds to 2 decimal places

```
FROM genre g
JOIN book b ON g.genre_ID = b.genre_ID
JOIN loan_detail ld ON b.book_id = ld.book_id
```

```
    JOIN loan l ON ld.loan_id = l.loan_id
```

-Joins the genre, book, loan_detail, and loan tables

-Connects genres to books, books to loan details, and loan details to actual loans

```
    GROUP BY g.genre_ID, g.book_name
)
```

-Groups the results by genre ID and genre name

-Required when using aggregate functions like COUNT() and SUM()

Final:

```
SELECT
    genre_ID,
    genre_name,
    total_loans,
    unique_books_loaned,
    unique_borrowers,
```

-Selects each genre's stats: ID, name, total loans, distinct books loaned, and borrowers

```
    percentage_of_total || '%' AS percentage_of_total_loans,
```

-Appends a % symbol to the percentage value for display

-Labels it as percentage_of_total_loans in the result

```
DENSE_RANK() OVER (ORDER BY total_loans DESC) AS popularity_rank
```

-Assigns a rank to each genre based on the number of loans

-Uses `DENSE_RANK` to give the same rank to genres with equal loan counts

```
FROM genre_loan_stats
```

-Refers to the genre_loan_stats CTE defined earlier

```
ORDER BY total_loans DESC
```

-Sorts genres in descending order of total loans

-Most borrowed genres appear first

```
LIMIT 3;
```

-Limits the final result to the **top 3 genres** based on total loans

```
tc3l-# LIMIT 3;
 genre_id |    genre_name    | total_loans | unique_books_loaned | unique_borrowers | percentage_of_total_loans | popularity_rank
----------+------------------+-------------+---------------------+------------------+---------------------------+-----------------
 G01      | Engineering      |           1 |                   1 |                1 | 25.00%                    |               1
 G06      | Computer Science |           1 |                   1 |                1 | 25.00%                    |               1
 G10      | Finance          |           1 |                   1 |                1 | 25.00%                    |               1
(3 rows)
```

Query 1: Display Top N Most Popular Books with Ranking and Reservation Analysis

Purpose:

This query identifies the most popular books in the library by analyzing loan frequency, reservation patterns, and availability ratios. It uses advanced window functions and subqueries to rank books by popularity and provides insights for library management decisions.

SQL Query:

```
WITH book_popularity AS (
   SELECT
      b.book_id,
      b.title,
      b.author,
      g.book_name AS genre,
      br.branch_name,
      b.total_copies,
      b.available_copies,
      COUNT(DISTINCT ld.loan_id) AS total_loans,
      COUNT(DISTINCT CASE
         WHEN r.book_status = 'reserved'
         THEN r.reservation_id
         END) AS active_reservations,
```

```
        ROUND(
            (b.available_copies * 100.0 / NULLIF(b.total_copies, 0)),
            1
        ) AS availability_percentage,
        ROUND(
            (COUNT(DISTINCT ld.loan_id) + COUNT(DISTINCT r.reservation_id) * 1.5)
            / NULLIF(b.total_copies, 0),
            2
        ) AS popularity_score
    FROM book b
    LEFT JOIN loan_detail ld ON b.book_id = ld.book_id
    LEFT JOIN reservation r ON b.book_id = r.book_id
    LEFT JOIN genre g ON b.genre_ID = g.genre_ID
    LEFT JOIN branch br ON b.branch_id = br.branch_id
    GROUP BY
        b.book_id, b.title, b.author, g.book_name,
        br.branch_name, b.total_copies, b.available_copies
)
SELECT
Book_id,title,author,genre,branch_name,total_copies,available_copies,total_loans,active_reservatio
ns,availability_percentage,popularity_score,
    RANK() OVER (ORDER BY popularity_score DESC) AS popularity_rank,
  CASE
        WHEN popularity_score >= 2.0 THEN 'Very High Demand'
        WHEN popularity_score >= 1.0 THEN 'High Demand'
        WHEN popularity_score >= 0.5 THEN 'Moderate Demand'
        ELSE 'Low Demand'
```

```
    END AS demand_level
FROM book_popularity
WHERE total_loans > 0 OR active_reservations > 0  -- Only show books with activity
ORDER BY popularity_score DESC
LIMIT 5;
```

line-by-line explanation:

CTE Declaration

```
WITH book_popularity AS (
```

- Creates a Common Table Expression (CTE) named book_popularity
- Allows complex calculations to be performed first, then referenced in the main query
- Acts as a temporary named result set that exists only for this query

Basic Book Information Selection

```
SELECT
    b.book_id,
    b.title,
    b.author,
    g.book_name AS genre,
    br.branch_name,
    b.total_copies,
    b.available_copies,
```

- Selects basic book identification: book_id, title, author
- Gets genre name from the genre table and aliases it as genre
- Retrieves branch_name to show which library branch houses the book

- Includes total_copies and available_copies for inventory tracking

Loan Count Calculation

```
COUNT(DISTINCT ld.loan_id) AS total_loans,
```

- Counts unique loan transactions for each book
- Uses DISTINCT to avoid double-counting if there are duplicate records
- Shows how many times each book has been borrowed historically

Active Reservations Count

```
COUNT(DISTINCT CASE
    WHEN r.book_status = 'reserved'
    THEN r.reservation_id
END) AS active_reservations,
```

- Counts only active reservations (status = 'reserved')
- Uses CASE WHEN to conditionally count only valid reservations
- DISTINCT ensures each reservation is counted once
- Shows current demand for unavailable books

Availability Percentage Calculation

```
ROUND(
    (b.available_copies * 100.0 / NULLIF(b.total_copies, 0)),
    1
) AS availability_percentage,
```

- Calculates what percentage of copies are currently available

- Formula: (available_copies × 100) ÷ total_copies
- NULLIF(b.total_copies, 0) prevents division by zero errors
- ROUND(..., 1) formats result to 1 decimal place
- Multiplies by 100.0 to convert to percentage format

Popularity Score Calculation

```
ROUND(
   (COUNT(DISTINCT ld.loan_id) + COUNT(DISTINCT r.reservation_id) * 1.5)
   / NULLIF(b.total_copies, 0),
   2
) AS popularity_score
```

- Creates a weighted popularity metric combining loans and reservations
- Formula: (total_loans + (total_reservations × 1.5)) ÷ total_copies
- Reservations weighted at 1.5× because they indicate higher demand
- Normalizes by total copies to account for inventory differences
- ROUND(..., 2) formats to 2 decimal places for precision

Table Joins

```
FROM book b
LEFT JOIN loan_detail ld ON b.book_id = ld.book_id
LEFT JOIN reservation r ON b.book_id = r.book_id
LEFT JOIN genre g ON b.genre_ID = g.genre_ID
LEFT JOIN branch br ON b.branch_id = br.branch_id
```

- Starts with book table as the base (aliased as b)
- LEFT JOIN loan_detail: Gets all borrowing history, keeps books with no loans
- LEFT JOIN reservation: Gets reservation data, keeps books with no reservations
- LEFT JOIN genre: Gets genre names for classification

- LEFT JOIN branch: Gets branch location information
- All LEFT JOINs ensure no books are excluded from results

Grouping for Aggregation

```
GROUP BY
    b.book_id, b.title, b.author, g.book_name,
    br.branch_name, b.total_copies, b.available_copies
```

- Groups results by book-level attributes to enable aggregation functions
- Each book appears once in results with calculated metrics
- All non-aggregated columns must appear in GROUP BY clause

Main Query Selection

```
SELECT
    book_id, title, author, genre, branch_name, total_copies,
    available_copies, total_loans, active_reservations,
    availability_percentage, popularity_score,
```

- Retrieves all calculated metrics from the CTE
- Includes both raw data and computed statistics

Ranking Calculation

```
RANK() OVER (ORDER BY popularity_score DESC) AS popularity_rank,
```

- Uses window function RANK() to assign positions based on popularity score

- ORDER BY popularity_score DESC ranks highest scores first
- RANK() handles ties properly (books with same score get same rank)
- Next rank after ties skips appropriate number (e.g., two books tied for 1st, next is 3rd)

Demand Level Classification

```
CASE
    WHEN popularity_score >= 2.0 THEN 'Very High Demand'
    WHEN popularity_score >= 1.0 THEN 'High Demand'
    WHEN popularity_score >= 0.5 THEN 'Moderate Demand'
    ELSE 'Low Demand'
END AS demand_level
```

- Creates categorical demand levels based on popularity score thresholds
- Uses nested CASE WHEN for multi-tier classification
- Provides business-friendly labels for interpretation
- Helps librarians quickly identify high-priority books

Data Source and Filtering

```
FROM book_popularity
WHERE total_loans > 0 OR active_reservations > 0
```

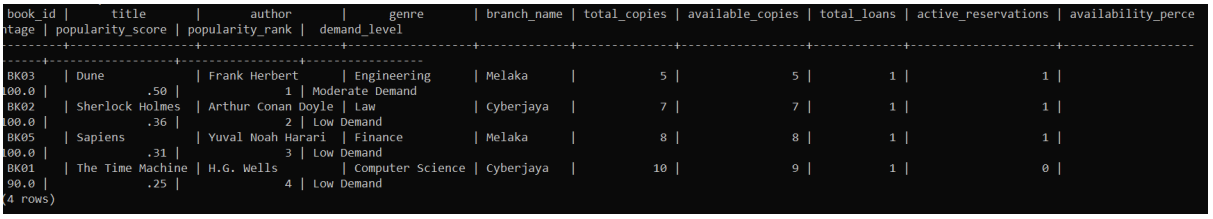
- References the CTE book_popularity as data source
- Filters to show only books with activity (loans OR reservations)
- Excludes books that have never been borrowed or reserved
- Focuses analysis on books with actual usage patterns

Final Sorting and Limiting

```
ORDER BY popularity_score DESC

LIMIT 5;
```

- Sorts results by popularity score in descending order (highest first)
- LIMIT 5 restricts output to top 5 most popular books
- Provides focused results for decision-making
- Can be adjusted to show different numbers of top books



```
book_id |      title       |       author       |      genre       | branch_name | total_copies | available_copies | total_loans | active_reservations | availability_perce
ntage | popularity_score | popularity_rank |  demand_level
--------+------------------+--------------------+------------------+-------------+--------------+------------------+-------------+---------------------+-------------------
------+------------------+-----------------+----------------
 BK03   | Dune             | Frank Herbert      | Engineering      | Melaka      |            5 |                5 |           1 |                   1 |
100.0 |             .50 |               1 | Moderate Demand
 BK02   | Sherlock Holmes  | Arthur Conan Doyle | Law              | Cyberjaya   |            7 |                7 |           1 |                   1 |
100.0 |             .36 |               2 | Low Demand
 BK05   | Sapiens          | Yuval Noah Harari  | Finance          | Melaka      |            8 |                8 |           1 |                   1 |
100.0 |             .31 |               3 | Low Demand
 BK01   | The Time Machine | H.G. Wells         | Computer Science | Cyberjaya   |           10 |                9 |           1 |                   0 |
 90.0 |             .25 |               4 | Low Demand
(4 rows)
```

Query 2: Pivot Table Simulation - Genre Popularity by Branch

SQL Query:

This query creates a pivot table showing genre popularity across two library branches (Cyberjaya and Melaka) with various statistics.

```
SELECT
    g.book_name AS genre,
    COUNT(CASE WHEN br.branch_name = 'Cyberjaya' THEN b.book_id END) AS
cyberjaya_books,
    COUNT(CASE WHEN br.branch_name = 'Melaka' THEN b.book_id END) AS melaka_books,
    SUM(CASE WHEN br.branch_name = 'Cyberjaya' THEN b.total_copies ELSE 0 END) AS
cyberjaya_total_copies,
    SUM(CASE WHEN br.branch_name = 'Melaka' THEN b.total_copies ELSE 0 END) AS
melaka_total_copies,
    ROUND(
        (SUM(CASE WHEN br.branch_name = 'Cyberjaya' THEN b.available_copies ELSE 0 END)
* 100.0) /
        NULLIF(SUM(CASE WHEN br.branch_name = 'Cyberjaya' THEN b.total_copies ELSE 0
END), 0), 2
```

```
  ) AS cyberjaya_availability_percent,
  ROUND(
    (SUM(CASE WHEN br.branch_name = 'Melaka' THEN b.available_copies ELSE 0 END) *
100.0) /
    NULLIF(SUM(CASE WHEN br.branch_name = 'Melaka' THEN b.total_copies ELSE 0
END), 0), 2
  ) AS melaka_availability_percent
FROM genre g
LEFT JOIN book b ON g.genre_ID = b.genre_ID
LEFT JOIN branch br ON b.branch_id = br.branch_id
GROUP BY g.book_name
ORDER BY (cyberjaya_books + melaka_books) DESC;
```

line-by-line explanation:

```
SELECT
  g.book_name AS genre,
```

- Selects the genre name from the genre table and aliases it as "genre"

```
  COUNT(CASE WHEN br.branch_name = 'Cyberjaya' THEN b.book_id END) AS
cyberjaya_books,
```

- Counts how many books exist in the Cyberjaya branch for each genre
- Uses CASE to only count when branch_name is 'Cyberjaya'

```
  COUNT(CASE WHEN br.branch_name = 'Melaka' THEN b.book_id END) AS melaka_books,
```

- Counts how many books exist in the Melaka branch for each genre
- Similar to above but for 'Melaka' branch

```
  SUM(CASE WHEN br.branch_name = 'Cyberjaya' THEN b.total_copies ELSE 0 END) AS
cyberjaya_total_copies,
```

- Sums the total copies of books in Melaka branch
- Same logic as above but for Melaka

```
  ROUND(
    (SUM(CASE WHEN br.branch_name = 'Cyberjaya' THEN b.available_copies ELSE 0 END)
```

```
* 100.0) /
      NULLIF(SUM(CASE WHEN br.branch_name = 'Cyberjaya' THEN b.total_copies ELSE 0
END), 0), 2
   ) AS cyberjaya_availability_percent,
```

- Calculates availability percentage for Cyberjaya:

  Sums available copies

  Divides by total copies (with NULLIF to prevent division by zero)

  Multiplies by 100 for percentage

  Rounds to 2 decimal places

```
   ROUND(
      (SUM(CASE WHEN br.branch_name = 'Melaka' THEN b.available_copies ELSE 0 END) *
100.0) /
      NULLIF(SUM(CASE WHEN br.branch_name = 'Melaka' THEN b.total_copies ELSE 0
END), 0), 2
   ) AS melaka_availability_percent
```

- Same availability calculation but for Melaka branch

```
FROM genre g
```

- Starts with the genre table (aliased as 'g') as our base

```
LEFT JOIN book b ON g.genre_ID = b.genre_ID
```

- Joins the book table to get all books for each genre
- LEFT JOIN ensures we keep genres even with no books

```
LEFT JOIN branch br ON b.branch_id = br.branch_id
```

- Joins the branch table to get branch information for each book
- Another LEFT JOIN to preserve all records

```
GROUP BY g.book_name
```

- Groups the results by genre name to get one row per genre

```
ORDER BY (cyberjaya_books + melaka_books) DESC;
```

- Orders results by total books across both branches
- DESC makes the most popular genres appear first

genre	cyberjaya_books	melaka_books	cyberjaya_total_copies	melaka_total_copies	cyberjaya_availability_percent	melaka_availability_percent
Business	0	1	0	7		100.00
Accounting	1	0	6	0	100.00	
Engineering	0	1	0	5		100.00
Law	1	0	7	0	100.00	
Multimedia	0	1	0	5		100.00
Finance	0	1	0	8		100.00
Marketing	1	0	9	0	100.00	
IT	0	1	0	5		100.00
Communication	1	0	4	0	100.00	
Computer Science	1	0	10	0	90.00	
Cinematic Arts	0	0	0	0		

(11 rows)

Contribution :

NAME	PERCENTAGE (%)
SITI ZULAIKHA BINTI ABDUL RAZIF	100
LAMA M. R. SIAM	100
EBA MOHAMED ABBAS AHMED	100
ESHIKA PRASAAD	100

Email

1. Zulaikha: SITI.ZULAIKHA.ABDUL1@student.mmu.edu.my
2. Lama:LAMA.M.R@student.mmu.edu.my
3. Eba: EBA.MOHAMED.ABBAS@student.mmu.edu.my
4. Eshika: ESHIKA.PRASAAD@student.mmu.edu.my