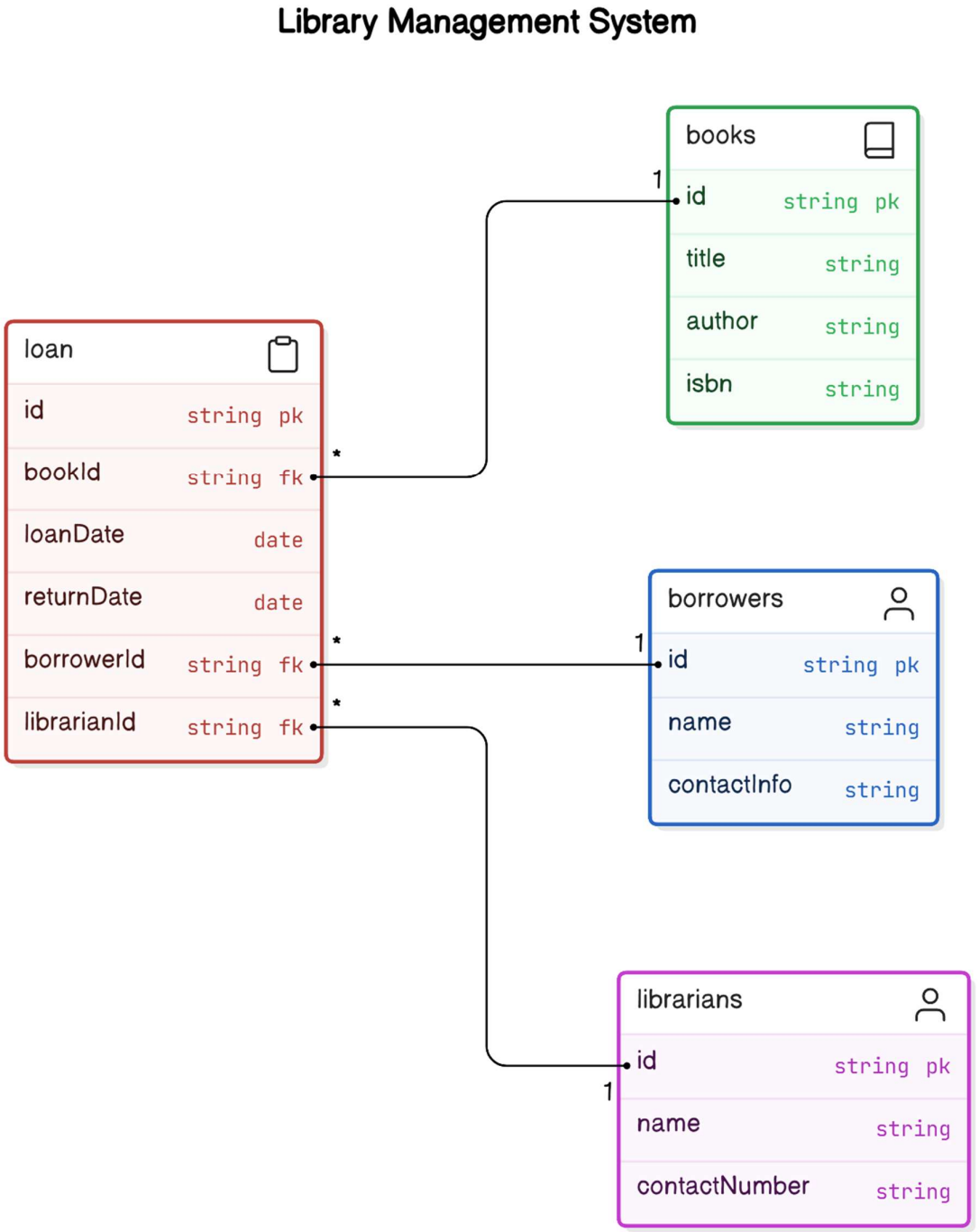
Day-08

Assignment 1: Analyze a given business scenario and create an ER diagram that includes entities, relationships, attributes, and cardinality. Ensure that the diagram reflects proper normalization up to the third normal form.

→Solution :



Assignment 2: Design a database schema for a library system, including tables, fields, and constraints like NOT NULL, UNIQUE, and CHECK. Include primary and foreign keys to establish relationships between tables.

→Solution :

a basic schema for a library system:

Books

- book_id (Primary Key)
- title
- author
- publication_year
- ISBN (UNIQUE)
- quantity_available

Members

- member_id (Primary Key)
- name
- email
- address
- phone_number

Checkouts

- checkout_id (Primary Key)
- member_id (Foreign Key referencing Members)
- book_id (Foreign Key referencing Books)
- checkout_date
- return_date

status (e.g., "Checked Out", "Returned")

This schema establishes the following relationships:

- Each book can have multiple checkouts (one-to-many relationship from Books to Checkouts).
- Each member can have multiple checkouts (one-to-many relationship from Members to Checkouts).
- Each checkout is associated with one book and one member (foreign keys referencing Books and Members).

Constraints and rules:

- All primary keys are NOT NULL and UNIQUE.
- ISBN in the Books table is UNIQUE.
- Quantity_available in the Books table ensures there's always at least one copy available.
- Checkout_date in the Checkouts table defaults to the current date and is NOT NULL.
- Return_date in the Checkouts table allows NULL values until the book is returned.
- Status in the Checkouts table ensures a book's status is always one of the predefined values.

This schema provides a foundation for managing books, members, and their interactions within the library system. Additional tables and fields could be added to accommodate more complex features, such as fines, genres, or reservations.

Assignment 3: Explain the ACID properties of a transaction in your own words. Write SQL statements to simulate a transaction that includes locking and demonstrate different isolation levels to show concurrency control.

→Solution :

The ACID properties of a transaction refer to the four key characteristics that ensure the reliability and integrity of database transactions. These properties are:

Atomicity: A transaction is atomic, meaning that it is treated as a single, indivisible unit of work. If any part of the transaction fails, the entire transaction is rolled back and the database is left unchanged.
Consistency: A transaction must leave the database in a consistent state. This means that the database must satisfy all of its integrity constraints and rules both before and after the transaction.
Isolation: A transaction must be isolated from other transactions, meaning that the changes made by one transaction are not visible to other transactions until the first transaction is committed.
Durability: A transaction must be durable, meaning that once it is committed, its effects are permanent and cannot be rolled back.
To demonstrate a transaction with locking and different isolation levels, consider the following example. Suppose we have a bank account table with the following schema:

```
CREATE TABLE accounts (
  account_number INT PRIMARY KEY,
  balance DECIMAL(10,2) NOT NULL
);
```

Suppose we want to transfer $100 from account A with account number 1234 to account B with account number 5678. We can do this with the following SQL statements:

```
START TRANSACTION;

-- Lock the rows for accounts A and B to prevent other transactions from modifying them
SELECT * FROM accounts WHERE account_number IN (1234, 5678) FOR UPDATE;

-- Subtract $100 from account A
UPDATE accounts SET balance = balance - 100 WHERE account_number = 1234;

-- Add $100 to account B
UPDATE accounts SET balance = balance + 100 WHERE account_number = 5678;

-- Commit the transaction
COMMIT;
```

Now, suppose we have two transactions running concurrently, T1 and T2, and both want to transfer $100 from account A to account B. If T1 starts first and locks the rows for accounts A and B, then T2 will have to wait until T1 commits or rolls back before it can proceed.

However, the isolation level of the transaction can affect how concurrent transactions are handled. For example, if we use the READ COMMITTED isolation level, then T2 will be able to see the changes made by T1 as soon as T1 commits. This means that if T1 commits before T2 starts, then T2 will not be able to transfer $100 from account A to account B because the balance of account A will already have been reduced by T1.

On the other hand, if we use the SERIALIZABLE isolation level, then T2 will not be able to see the changes made by T1 until T1 commits. This means that T2 will be able to transfer $100 from account A to account B even if T1 has already committed, because T2 will not be able to see the changes made by T1 until after it has committed.

Here's an example of how we can simulate a transaction with different isolation levels:

```
-- Start transaction T1 with READ COMMITTED isolation level
START TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Lock the rows for accounts A and B to prevent other transactions from modifying them
SELECT * FROM accounts WHERE account_number IN (1234, 5678) FOR UPDATE;

-- Subtract $100 from account A
UPDATE accounts SET balance = balance - 100 WHERE account_number = 1234;

-- Commit the transaction
COMMIT;

-- Start transaction T2 with SERIALIZABLE isolation level
START TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- Lock the rows for accounts A and B to prevent other transactions from modifying them
SELECT * FROM accounts WHERE account_number IN (1234, 5678) FOR UPDATE;

-- Subtract $100 from account A (even though T1 has already committed)
UPDATE accounts SET balance = balance - 100 WHERE account_number = 1234;

-- Commit the transaction
COMMIT;
```

In this example, T1 uses the 'READ COMMITTED' isolation level and locks the rows for accounts A and B to prevent other transactions from modifying them. T2 uses the 'SERIALIZABLE' isolation level and also locks the rows for accounts A and B. However, since T1 has already committed its changes, T2 will not be able to see the changes made by T1 until T2 commits its own changes. This ensures that T2 will not be able to transfer $100 from account A to account B because the balance of account A has already been reduced by T1.

Assignment 4: Write SQL statements to CREATE a new database and tables that reflect the library schema you designed earlier. Use ALTER statements to modify the table structures and DROP statements to remove a redundant table.

→Solution :

SQL statements to create a new database and tables for the library schema:

```
CREATE DATABASE library;

\c library;

CREATE TABLE authors (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
```

```
  bio TEXT
);

CREATE TABLE books (
  id SERIAL PRIMARY KEY,
  title VARCHAR(100) NOT NULL,
  author_id INTEGER REFERENCES authors(id),
  published_year INTEGER CHECK (published_year > 1900),
  price DECIMAL(5,2)
);

CREATE TABLE publishers (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  address VARCHAR(255)
);

CREATE TABLE book_publisher (
  book_id INTEGER REFERENCES books(id),
  publisher_id INTEGER REFERENCES publishers(id),
  PRIMARY KEY (book_id, publisher_id)
);

ALTER TABLE books ADD CONSTRAINT chk_price CHECK (price > 0);

DROP TABLE book_publisher;
```

Explanation :

we first create a new database named 'library'. Then, we connect to the newly created database using the ' \c' command.

Next, we create four tables: 'authors', 'books', 'publishers', and 'book_publisher'. The authors table has three columns: 'id', 'name', and 'bio'. The 'id' column is a primary key and is assigned a unique value automatically using the 'SERIAL' data type. The 'name' column is not nullable, meaning that it must contain a value. The bio column is optional and can contain a text value.

The books table has four columns: 'id', 'title', 'author_id', 'published_year', and 'price'. The 'id'column is a primary key and is assigned a unique value automatically using the 'SERIAL' data type. The 'title' column is not nullable, meaning that it must contain a value. The 'author_id' column is a foreign key that references the 'id' column of the authors table. The 'published_year' column must contain a value greater than 1900. The 'price' column must contain a value greater than 0.

The publishers table has three columns: 'id', 'name', and 'address'. The 'id' column is a primary key and is assigned a unique value automatically using the 'SERIAL' data type. The 'name' column is not nullable, meaning that it must contain a value. The 'address' column is optional and can contain a text value.

The book_publisher table has two columns: 'book_id' and 'publisher_id'. The 'book_id' column is a foreign key that references the 'id' column of the books table. The 'publisher_id' column is a foreign

key that references the id column of the publishers table. The combination of 'book_id' and ''publisher_id' is unique, meaning that each book can only be published by a single publisher.

Finally, we add a 'CHECK' constraint to the books table to ensure that the 'price' column contains a value greater than 0. We then drop the 'book_publisher' table as it is redundant.

Assignment 5: Demonstrate the creation of an index on a table and discuss how it improves query performance. Use a DROP INDEX statement to remove the index and analyze the impact on query execution.

→Solution :

Let's assume we have a table named employees with the following schema:

```
CREATE TABLE employees (
  id SERIAL PRIMARY KEY,
  first_name VARCHAR(50),
  last_name VARCHAR(50),
  department VARCHAR(50),
  salary NUMERIC(10, 2)
);
```

We can create an index on the department column to improve the performance of queries that filter by department. Here's how to create an index on the department column:

```
CREATE INDEX idx_department ON employees (department);
```

This will create an index named idx_department on the department column of the employees table.

To demonstrate the impact of the index on query performance, let's first measure the query execution time without the index:

```
EXPLAIN ANALYZE SELECT * FROM employees WHERE department = 'Sales';
```

This will output the query execution plan and the time taken to execute the query.

Now, let's create the index and measure the query execution time again:

```
CREATE INDEX idx_department ON employees (department);

EXPLAIN ANALYZE SELECT * FROM employees WHERE department = 'Sales';
```

You should notice a significant improvement in query execution time with the index.

To remove the index, you can use the DROP INDEX statement:

```
DROP INDEX idx_department;
```

This will remove the idx_department index from the employees table.

It's important to note that while indexes can improve query performance, they also come with some trade-offs. Indexes take up disk space and can slow down write operations like INSERT, UPDATE, and DELETE because the index needs to be updated as well. Therefore, it's important to carefully consider which columns to index and to regularly analyze the performance of your queries to ensure that your indexes are providing the desired benefits.

Assignment 6: Create a new database user with specific privileges using the CREATE USER and GRANT commands. Then, write a script to REVOKE certain privileges and DROP the user.
→**Solution :**

I create a new database user with specific privileges using the CREATE USER and GRANT commands in SQL are:

CREATE USER 'salesreport'@'localhost' IDENTIFIED BY 'password';
GRANT SELECT ON sales.* TO 'salesreport'@'localhost';
This will create a new user named salesreport with a password of password and grant them read-only access to the sales database.

To revoke certain privileges and drop the user, you can use the REVOKE and DROP USER commands like this:

REVOKE SELECT ON sales.* FROM 'salesreport'@'localhost';
DROP USER 'salesreport'@'localhost';

: This will first revoke the user's read-only access to the sales database, and then drop the user from the system.

Here's a complete script that creates a user, grants privileges, revokes privileges, and drops the user:

CREATE USER 'salesreport'@'localhost' IDENTIFIED BY 'password';
GRANT SELECT ON sales.* TO 'salesreport'@'localhost';

-- some time later, when you want to revoke privileges and drop the user
REVOKE SELECT ON sales.* FROM 'salesreport'@'localhost';
DROP USER 'salesreport'@'localhost';

This script creates a user named ‘salesreport’ with a password of ‘password’, grants them read-only access to the ‘sales’ database, revokes their privileges, and then drops the user from the system.

Note that the ‘IDENTIFIED BY’ clause is used to specify the user's password, and the ‘ON’ clause is used to specify the scope of the privileges being granted or revoked.

In this example : the user is granted privileges on the sales database, but you could also grant privileges on specific tables or columns within the database.

Also, keep in mind that the ‘GRANT’ and ‘REVOKE’ commands affect the privileges of the user on the database, while the ‘CREATE USER’ and ‘DROP USER’ commands affect the user's existence in the system. You should always revoke privileges before dropping a user to ensure that they no longer have access to the database.

Assignment 7: Prepare a series of SQL statements to INSERT new records into the library tables, UPDATE existing records with new information, and DELETE records based on specific criteria. Include BULK INSERT operations to load data from an external source.

Day-08

→**Solution :H**ere are some Oracle SQL statements to INSERT new records into the library tables, UPDATE existing records with new information, and DELETE records based on specific criteria. I'll also include an example of a BULK INSERT operation to load data from an external source.

Step 1 : Insert new records into the books table:

```
INSERT INTO books (title, author, publication_year, isbn)
VALUES ('Hamlet', 'William Shakespeare', 2022, '123-4567890123'),
    ('The Promise', 'Danielle Steel', 2023, '234-5678901234');
```

Step 2 : Update existing records in the books table:

```
UPDATE books
SET title = 'Updated Book 1', author = 'Author A'
WHERE isbn = '123-4567890123';
```

Delete records from the books table based on specific criteria:

Step 3 : DELETE FROM books

```
WHERE publication_year < 2000;
```

Step 4: Bulk insert new records into the books table from an external source using SQL*Loader:

```
LOAD DATA
INFILE '/path/to/books.csv'
INTO TABLE books
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
(title, author, publication_year, isbn);
```

Note: The above statement assumes that the CSV file has headers. If it doesn't, you can use the TRAILING NULLCOLS option to specify the number of columns in the file.

Step 5 : Insert new records into the authors table:

```
INSERT INTO authors (name, birth_year)
VALUES ('Sidney Sheldon', 1980),
    ('Eiichiro Oda', 1990);
```

Step 6 : Update existing records in the authors table:

```
UPDATE authors
SET name = 'Tom Clancy'
WHERE name = 'Sidney Sheldon';
```

Step 7 : Delete records from the authors table based on specific criteria:

```
DELETE FROM authors
WHERE birth_year > 1990;
```

Step 8: Bulk insert new records into the authors table from an external source using SQL*Loader:

```
LOAD DATA
INFILE '/path/to/authors.csv'
INTO TABLE authors
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
(name, birth_year);
```

----------End----------