E-BOOK PROJECT

ADVANCED DATABASE SYSTEM

Reg. No: UGC0122015

Name : Oneli Jayodya

Batch : SE 01

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 1 : INTRODUCTION

1.1. Introduction

This project is an Online Book purchase. In this chapter, you can learn the idea of this report and what will be discussed in this report. First, introduce our system and why chose this system to build, and then we will discuss the project's scope, objectives, and functionalities.

1.2. Introduction of the system

E-book reading is making great strides, and it has become imperative to introduce efficient E-Book management systems. The database project is Book Heaven. This proposes a database system for Book Heaven where users can purchase, and manage books in different languages. Book Heaven can be enriched with books from children to elders. In this project, the main concern will be to design and develop a PostgreSQL-based database that manages all necessary data about the platform. We will implement a database for data manipulation and handling with the help of pgAdmin.

The project aims to develop a comprehensive database-driven management system for a bookstore. It will streamline operations, manage inventory, handle customer transactions, and provide automated processes for payment, book availability, and author management.

1.3. Scope

The scope of this proposal is limited to the design and implementation of a PostgreSQL database for Book Heaven. The database will be built to manage the core functionalities of the platform. Specifically, the database will handle:

- **Books Management**: Add, update, and delete books
- **Author Management**: Insert, update, and manage author details, ensuring unique names.
- **Customer Book Cart**: Add books to a customer's cart while verifying availability. Automatically update payment amounts based on book cart changes.
- **Payment System**: Maintain a running total of customer purchases. Automatically update the payment table when books are added/removed from carts or deleted from inventory.

1.4. Functionalities

In Book Heaven, a well-designed PostgreSQL database can generate a variety of reports to help manage the platform, track user activity, and analyze book performance. Here are some key reports that would be valuable:

1. Sales Reports – (Daily/Monthly/Yearly Sales, Top-Selling Books)
2. User Activity Reports – (Purchase Frequency)
3. Book Performance Reports – (Highest Rated Books, most time purchased book)
4. Book Language and Type – (In each type how many books in English)
5. Available books Report

1.5. Objectives

The objectives of this database implementation are as follows:

1. **To efficiently manage user and book data**: Develop a relational database structure that handles user accounts, book information, reviews, and purchases.
2. **To facilitate scalability**: The database will be designed to easily accommodate an increasing number of books, users, and transactions without compromising performance.
3. **To provide multilingual support**: Ensure that the database structure is capable of storing and retrieving books in different languages, making the platform accessible to a diverse user base.
4. **Query and Reporting Capability:** Enable users to query and retrieve recommendations efficiently.
5. **Data Integrity:** Use triggers, stored procedures, and functions to ensure consistent and accurate data management.
6. **Error Handling:** Implement mechanisms for handling constraints, such as book availability and unique author names, with meaningful feedback.

1.6. Summary

This chapter introduces the Book Heaven project, an online book-purchasing system focused on managing books, authors, customers, and transactions efficiently. The chapter highlights the project's purpose, scope, and core functionalities, providing a clear understanding of the system's objectives and benefits. In next chapter outlines the database schema and reports queries, transactions, indexing, procedures, functions, and triggers.

Chapter 2 : DATABASE DESIGN

2.1. Introduction

In the previous chapter, we discussed the system and its scope, what are its functionality, reports that can be made through this, and objectives. In this chapter, we will discuss tables in this e-book system and report queries, indexes, functions and triggers, and procedures in this system.

2.2. Tables

The database is book_heaven. You can see how to create tables and insert data into the database. There are 6 tables which are languages, authors, customers, books, book cart, payment, and paid. Here are the tables;

Query | Query History

```
1  -- authors table
2  CREATE TABLE authors (
3      id SERIAL PRIMARY KEY,
4      name VARCHAR(50) NOT NULL UNIQUE,
5      gender VARCHAR(10),
6      city VARCHAR(25)
7  );
```

Figure 2.2.1: Author table

The author table is designed to store information about book authors. Each author is identified by a unique ID. The table includes fields for the author's name, gender, and city, ensuring basic personal and location details are recorded. (see Figure 2.2.1)

```sql
--languages table
CREATE TABLE languages (
    id SERIAL PRIMARY KEY,
    language VARCHAR(30) NOT NULL UNIQUE
);
```

Figure 2.2.2: language table

The languages table is designed to manage the list of languages available for the books in the system. Each language is identified by a unique ID, and the language field ensures that each entry represents a distinct language. (see Figure 2.2.2)

```sql
-- customers table
CREATE TABLE customers (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    gender VARCHAR(10),
    age INT CHECK (age > 0),
    city VARCHAR(25)
);
```

Figure 2.2.3: customer table

The customer table stores information about the users who interact with the system. Each customer is uniquely identified by an ID and includes attributes such as name, gender, age (must be greater than 0), and city. (see Figure 2.2.3)

```sql
-- books table
CREATE TABLE books (
    id SERIAL PRIMARY KEY,
    title VARCHAR(100) NOT NULL UNIQUE,
    price NUMERIC(10, 2) NOT NULL CHECK (price > 0),
    authorid INT NOT NULL,
    languageid INT NOT NULL,
    availability BOOLEAN DEFAULT TRUE,
    FOREIGN KEY (authorid) REFERENCES authors (id),
    FOREIGN KEY (languageid) REFERENCES languages (id)
);
```

Figure 2.2.4: books table

The books table serves as the central repository for storing information about the books available in the system. Each book is identified by a unique ID and includes details such as title, price (must be greater than 0), and availability (indicating if the book is currently available). The table is linked to the authors and languages tables via foreign keys, authorized, and language, ensuring that each book is associated with an existing author and language. (see Figure 2.2.4)

```sql
-- bookcart table
CREATE TABLE bookcart (
    id SERIAL PRIMARY KEY,
    customerid INT NOT NULL,
    bookid INT NOT NULL,
    amount INT NOT NULL CHECK(amount>0),
    FOREIGN KEY (customerid) REFERENCES customers (id),
    FOREIGN KEY (bookid) REFERENCES books (id)
);
```

Figure 2.2.5: book cart table

The book table stores the information about the books added to each customer's cart. Each record is uniquely identified by an ID and includes references to the customer ID and the book. The amount field indicates the quantity of each book the customer intends to purchase, with a check that the amount is greater than 0. (see Figure 2.2.5)

```sql
-- payment table
CREATE TABLE payment (
    id SERIAL PRIMARY KEY,
    customerid INT NOT NULL,
    totalamount NUMERIC(10, 2) NOT NULL CHECK (totalamount >= 0),
    payment_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP, -- Automatically sets the current date
    FOREIGN KEY (customerid) REFERENCES customers (id)
);
```

Figure 2.2.6: payment table

The payment table records the payment details for customers. It includes the following fields id, customerid, total amount, and payment_date. The timestamp when the payment was made, is automatically set to the current date and time when a new record is inserted. This table is important for tracking customer transactions and maintaining accurate records of payments in the system.

```sql
--paid table
CREATE TABLE paid (
    id SERIAL PRIMARY KEY,
    customerid INT NOT NULL,
    totalprice NUMERIC(10, 2) NOT NULL CHECK (totalprice >= 0),
    paid_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (customerid) REFERENCES customers (id)
);
```

Figure 2.2.7: paid table

The paid table records details about payments that have been successfully processed. It includes the following fields id, customerid, total price, and paid_date. This table serves as a record of completed transactions, providing a history of payments made by customers in the system. (see Figure 2.2.7)

2.3. Queries

In the book_heaven database, there are 6 tables. They are users, books, authors, languages, categories, transactions. In the below, you can see some queries that help to make decisions on your system.

```sql
SELECT DATE(paid_date) AS sale_date, SUM(totalprice) AS total_sales
FROM paid
GROUP BY DATE(paid_date)
ORDER BY sale_date;
```

Data Output | Messages | Notifications

	sale_date date	total_sales numeric
1	2024-12-27	22700.00
2	2024-12-29	1400.00

Figure 2.3.1: query 01

This query retrieves the daily sales of the book heaven. It calculates the total revenue and number of transactions for the current day, helping track daily sales performance. (see Figure 2.3.1)

```
931 SELECT b.title, SUM(bc.amount) AS total_sold
932 FROM bookcart bc
933 JOIN books b ON bc.bookid = b.id
934 GROUP BY b.title
935 ORDER BY total_sold DESC
936 LIMIT 5;
```

Data Output | Messages | Notifications

	title character varying (100)	total_sold bigint
1	The Heart of the Island	14
2	Fading Memories	12
3	The Forbidden Path	10
4	The Storm Within	9
5	Beneath the Moonlit Sky	9

Figure 2.3.2: query 02

This SQL query retrieves the titles of the 5 least borrowed books along with the number of times each book was borrowed. This report shows the top 5 best-selling books, helping identify popular titles for restocking or marketing. (see Figure 2.3.2)

```
945 SELECT p.customerid, SUM(p.totalamount) AS total_spent
946 FROM payment p
947 GROUP BY p.customerid
948 ORDER BY total_spent DESC;
949
```

Data Output | Messages | Notifications

	customerid integer	total_spent numeric
1	4	12400.00
2	11	11400.00
3	9	10700.00
4	6	10700.00
5	5	10300.00

Figure 2.3.3: query 03

This query provides a summary of total spending by each customer, which helps in identifying high-value customers. (see Figure 2.3.3)

```
939 SELECT DISTINCT c.id, c.name
940 FROM customers c
941 JOIN bookcart bc ON c.id = bc.customerid
942 WHERE bc.bookid IS NOT NULL;
943
```

Data Output | Messages | Notifications

	id [PK] integer	name character varying (50)
1	6	Tharindu Weerasinghe
2	20	Samitha Gunasekara
3	15	Chathuri Fernando
4	4	Chandana Silva
5	16	Buddhika Senanayake
6	25	Nimesh Bandara
7	19	Dinesh Bandara
8	7	Lakmini Gunawardena
9	23	Lakshitha Perera
10	18	Ruvini Silva

Figure 2.3.4: query 04

This query identifies customers who are actively browsing or adding books to their cart, useful for engagement or promotional targeting. (see Figure 2.3.4)

```
951 SELECT b.title, b.price, b.availability
952 FROM books b
953 WHERE b.availability = FALSE;
954
```

Data Output | Messages | Notifications

	title character varying (100)	price numeric (10,2)	availability boolean
1	The Sea of Silence	1600.00	false
2	The Path of the Warrior	1800.00	false
3	Waves of Time	1500.00	false
4	Heart	1700.00	false
5	Tides of Change	1400.00	false
6	The Roar of the Tiger	1600.00	false

Figure 2.3.5: query 05

This Identifies books that are out of stock, which are not available currently allowing the inventory team to act quickly and restock popular titles. (see Figure 2.3.5)

```
956 SELECT a.name AS author_name, b.title, b.price
957 FROM books b
958 JOIN authors a ON b.authorid = a.id
959 WHERE a.name = 'Amal Perera';
960
```

Data Output | Messages | Notifications

	author_name character varying (50)	title character varying (100)	price numeric (10,2)
1	Amal Perera	The Silence of the Sea	1500.00
2	Amal Perera	Beyond the Boundaries	1500.00
3	Amal Perera	Madol Doova	1500.00
4	Amal Perera	Kaliyugaya	1600.00
5	Amal Perera	Gamperaliya	1700.00
6	Amal Perera	The Village in the Jungle	1500.00

Figure 2.3.6: query 06

This query provides a list of all books by a specific author, which can be used to track an author's portfolio or promote their work. (see Figure 2.3.6)

```
962 SELECT l.language, SUM(bc.amount * b.price) AS total_sales
963 FROM bookcart bc
964 JOIN books b ON bc.bookid = b.id
965 JOIN languages l ON b.languageid = l.id
966 GROUP BY l.language;
967
```

Data Output | Messages | Notifications

	language character varying (30)	total_sales numeric
1	Tamil	51500.00
2	English	50700.00
3	Japanese	56500.00
4	Sinhala	54800.00

Figure 2.3.7: query 07

Tracks sales by language, which helps determine which language books are in demand in your bookstore. (see Figure 2.3.7)

```
969 SELECT a.name AS author_name, SUM(bc.amount * b.price) AS total_sales
970 FROM bookcart bc
971 JOIN books b ON bc.bookid = b.id
972 JOIN authors a ON b.authorid = a.id
973 GROUP BY a.name;
974
```

Data Output | Messages | Notifications

	author_name character varying (50)	total_sales numeric
1	Amal Rathnayake	1500.00
2	Roshan Gunasekara	7200.00
3	Anura Aluthge	6800.00
4	Sulochana Kodithuwakku	9600.00
5	Sarath Tennakoon	11400.00
6	Menaka Rajapaksha	1700.00
7	Upul Dharmapala	4500.00
8	Rashmi Perera	5600.00
9	Anjali Perera	5400.00
10	Sunil Abeysekara	6000.00

Figure 2.3.8: query 08

This query helps to track total sales by author, providing insight into which authors are driving the most revenue. (see Figure 2.3.8)

```
976 SELECT p.customerid, SUM(p.totalamount) AS total_paid
977 FROM payment p
978 GROUP BY p.customerid;
979
```

Data Output | Messages | Notifications

	customerid integer	total_paid numeric
1	23	6800.00
2	24	7700.00
3	11	11400.00
4	8	5900.00
5	19	9800.00
6	25	7400.00
7	4	12400.00

Figure 2.3.9: query 09

Summarizes the total payment made by each customer, useful for customer segmentation or loyalty programs. (see Figure 2.3.9)

```
982 SELECT c.gender, COUNT(bc.bookid) AS total_books_bought, SUM(bc.amount * b.price) AS total_spent
983 FROM bookcart bc
984 JOIN books b ON bc.bookid = b.id
985 JOIN customers c ON bc.customerid = c.id
986 GROUP BY c.gender;
987
988
```

Data Output | Messages | Notifications

	gender character varying (10)	total_books_bought bigint	total_spent numeric
1	Female	36	106300.00
2	Male	37	107200.00

Figure 2.3.10: query 10

This query reveals which gender buys more books and how much they spend, providing insights into customer behavior for targeted marketing campaigns. (see Figure 2.3.10)

2.4. Indexing

This system uses two types of indexes secondary and multi-index. A secondary index is an additional data structure created on a database table to allow for faster retrieval of records based on non-primary key columns. Multi-indexing refers to creating multiple indexes on different columns of the same table or creating a composite index that spans multiple columns.

```
1200 EXPLAIN SELECT * FROM customers WHERE age = 25 AND city = 'Galle';
1201
1202
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on customers (cost=0.00..10.06 rows=1 width=38)
2	Filter: ((age = 25) AND ((city)::text = 'Galle'::text))

Figure 2.4.1:before index 1

In this explain query show customer details that are in age 25 and city is Galle. The cost is 10.06. It scans using sequence scan. Next, we add an index to the customer table. (see Figure 2.4.1)

```
1198  CREATE INDEX idx_age_city ON customers (age, city);
1199
1200  EXPLAIN SELECT * FROM customers WHERE age = 25 AND city = 'Galle';
1201
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Index Scan using idx_age_city on customers (cost=0.27..8.29 rows=1 width=38)
2	Index Cond: ((age = 25) AND ((city)::text = 'Galle'::text))

Figure 2.4.2: after indexing 1

This shows after adding an index to the customers. The index name is idx_age_city. This is a multi-index. Now the cost is 8.29. you can see it scanned using an index scan. And the cost is reduced. (see Figure 2.4.2)

```
1189  EXPLAIN SELECT * FROM authors WHERE name = 'J.K. Rowling';
1190
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Seq Scan on authors (cost=0.00..6.38 rows=1 width=35)
2	Filter: ((name)::text = 'J.K. Rowling'::text)

Figure 2.4.3: before index 2

In this explanation the authors detail where the name is equal to J.K. Rowling. The cost is 6.38. It scans using sequence scan. (see Figure 2.4.3)

```
1279  CREATE INDEX idx_author_name ON authors (name);
1280
1281  EXPLAIN SELECT * FROM authors WHERE name = 'J.K. Rowling';
1282
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	Index Scan using idx_author_name on authors (cost=0.27..8.29 rows=1 width=...
2	Index Cond: ((name)::text = 'J.K. Rowling'::text)

Figure 2.4.4: after index 2

This shows after adding an index to the customers. This is a secondary index. You can see it scanned using an index scan. (see Figure 2.4.4)

2.5. Functions and Triggers

Functions and triggers in databases are essential for automating tasks and enhancing data integrity. A function is a reusable block of code that performs a specific operation. On the other hand, a trigger is a special kind of stored procedure that is automatically executed in response to certain events on a particular table or view. Together, functions and triggers provide powerful ways to manage and maintain database operations.

```
1573  CREATE OR REPLACE FUNCTION check_book_availability()
1574  RETURNS TRIGGER AS $$
1575  BEGIN
1576      -- Check if the book is available
1577      IF NOT EXISTS (SELECT 1 FROM books WHERE id = NEW.bookid AND availability = TRUE) THEN
1578          RAISE EXCEPTION 'Book ID % is not available.', NEW.bookid;
1579      END IF;
1580
1581      RAISE NOTICE 'Book added to cart successfully.';
1582
1583      -- Allow the insert to proceed
1584      RETURN NEW;
1585  END;
1586  $$ LANGUAGE plpgsql;
1587
1588  CREATE TRIGGER before_insert_bookcart
1589  BEFORE INSERT ON bookcart
1590  FOR EACH ROW
1591  EXECUTE FUNCTION check_book_availability();
1592
1593  INSERT INTO bookcart (customerid, bookid, amount) VALUES (401, 906, 1);
1594
Data Output  Messages  Notifications

ERROR:  Book ID 906 is not available.
```

Figure 2.5.1: check book

In this function check the book availability before insert to the book cart. If the customer inserts their ID, book ID, and amount of buying that book then it first checks whether the book is available, if it does not then shows the error message. If the book is available then add it to the book cart table and show the book added to the cart successfully message. Use the trigger to check each row before inserting it to do it automatically. (see Figure 2.5.1)

```
1601 CREATE OR REPLACE FUNCTION calculate_total_amount()
1602 RETURNS TRIGGER AS $$
1603 DECLARE
1604     total NUMERIC(10, 2);
1605 BEGIN
1606     -- Calculate total amount for the customer
1607     SELECT SUM(b.price * bc.amount)
1608     INTO total
1609     FROM bookcart bc
1610     JOIN books b ON bc.bookid = b.id
1611     WHERE bc.customerid = NEW.customerid;
1612
1613     -- If the payment record exists, update it
1614     IF EXISTS (SELECT 1 FROM payment WHERE customerid = NEW.customerid) THEN
1615         UPDATE payment
1616         SET totalamount = total
1617         WHERE customerid = NEW.customerid;
1618     ELSE
1619         -- If the payment record doesn't exist, insert a new one
1620         INSERT INTO payment (customerid, totalamount)
1621         VALUES (NEW.customerid, total);
1622     END IF;
1623
1624     RETURN NEW;
1625 END;
1626 $$ LANGUAGE plpgsql;
1627
1628 CREATE TRIGGER update_payment_total
1629 AFTER INSERT OR UPDATE ON bookcart
1630 FOR EACH ROW
1631 EXECUTE FUNCTION calculate_total_amount();
```

Figure 2.5.2: total payment

The provided PostgreSQL code defines a function calculate_total_amount() in plpgSQL, which calculates the total price of items in a customer's cart by multiplying book prices and quantities from the bookcart table, joined with the books table. It checks whether an entry for the customer exists in the payment table: if it does, the total amount is updated; otherwise, a new record is inserted. The function is linked to a trigger update_payment_total that activates after every insert or update operation on the bookcart table to ensure the payment table reflects the latest cart details. (see Figure 2.5.2)

```
1821
1822  CREATE OR REPLACE FUNCTION get_sales_by_date()
1823  RETURNS TABLE (sale_date DATE, total_sales NUMERIC) AS $$
1824  BEGIN
1825      RETURN QUERY
1826      SELECT DATE(paid_date) AS sale_date, SUM(totalprice) AS total_sales
1827      FROM paid
1828      GROUP BY DATE(paid_date)
1829      ORDER BY sale_date;
1830  END;
1831  $$ LANGUAGE plpgsql;
1832
1833  SELECT * FROM get_sales_by_date();
1834
```

Data Output | Messages | Notifications

	sale_date date	total_sales numeric
1	2024-12-27	22700.00
2	2024-12-29	1400.00

Figure 2.5.3: daily sales

The get_sales_by_date function is a PostgreSQL user-defined function that calculates daily total sales by grouping records from the paid table based on the paid_date column. It returns a table with two columns: sale_date (the date of sales) and total_sales (the sum of totalprice for that date). The function uses SQL aggregation with GROUP BY and sorts the results by sale_date. It is designed for easy retrieval of summarized sales data by date. (see Figure 2.5.3)

2.6. Procedures

A procedure in PostgreSQL is a database object that contains a set of SQL and procedural commands to perform a specific task. Unlike functions, procedures can execute SQL commands that modify database schema or data and do not necessarily return a value. Procedures are invoked using the CALL statement.

```
1836  CREATE OR REPLACE PROCEDURE calculate_author_sales()
1837  LANGUAGE plpgsql AS $$
1838  DECLARE
1839      author_name TEXT;
1840      total_sales NUMERIC;
1841  BEGIN
1842      -- RAISE NOTICE for clarity
1843      RAISE NOTICE 'Calculating Author Sales...';
1844
1845      FOR author_name, total_sales IN
1846          SELECT a.name, SUM(bc.amount * b.price)
1847          FROM bookcart bc
1848          JOIN books b ON bc.bookid = b.id
1849          JOIN authors a ON b.authorid = a.id
1850          GROUP BY a.name
1851      LOOP
1852          RAISE NOTICE 'Author: % ==> Total: %', author_name, total_sales;
1853      END LOOP;
1854  END;
1855  $$;
1856
1857  CALL calculate_author_sales();
1858
```

Data Output | Messages | Notifications

```
NOTICE:  Calculating Author Sales...
NOTICE:  Author: Amal Rathnayake ==> Total: 3000.00
NOTICE:  Author: Himani Wijesinghe ==> Total: 2800.00
NOTICE:  Author: Roshan Gunasekara ==> Total: 14400.00
NOTICE:  Author: Sarath Tennakoon ==> Total: 26400.00
```

Figure 2.6.1: author sales

The calculate_author_sales procedure calculates each author's total sales by summing up the product of book prices and the number of books sold from the book cart table. The results are displayed using RAISE NOTICE, showing each author's name and corresponding total sales. This procedure is beneficial for automating repetitive tasks and can be executed with a simple CALL statement. (see Figure 2.6.1)

```
1795  CREATE OR REPLACE PROCEDURE delete_book(p_bookid INT)
1796  LANGUAGE plpgsql
1797  AS $$
1798  BEGIN
1799      -- Check if the book exists
1800      IF NOT EXISTS (SELECT 1 FROM books WHERE id = p_bookid) THEN
1801          RAISE NOTICE 'Book with ID % does not exist.', p_bookid;
1802          RETURN;
1803      END IF;
1804
1805      -- Optionally, delete related entries in the bookcart table
1806      DELETE FROM bookcart
1807      WHERE bookid = p_bookid;
1808
1809      -- Delete the book from the books table
1810      DELETE FROM books
1811      WHERE id = p_bookid;
1812
1813      RAISE NOTICE 'Book with ID % deleted successfully.', p_bookid;
1814  END;
1815  $$;
1816
1817  CALL delete_book(3);
```

Figure 2.6.2: delete book

The delete_book procedure is a PostgreSQL stored procedure designed to delete a book and its related entries from the database. It accepts a p_bookid parameter, which specifies the ID of the book to be deleted. (see Figure 2.6.2) The procedure first checks if the book exists in the books table. If the book does not exist, it raises a notice and exits. If the book exists, it deletes any related entries in the bookcart table to maintain referential integrity and removes the book from the book table. After successful deletion, shows a confirmation notice. The procedure is invoked using the CALL statement with the desired book ID.

```
1860 CREATE OR REPLACE PROCEDURE calculate_sales_by_language()
1861 LANGUAGE plpgsql
1862 AS $$
1863 DECLARE
1864     language TEXT;
1865     total_sales NUMERIC;
1866 BEGIN
1867
1868     RAISE NOTICE 'Sales by language:';
1869     FOR language, total_sales IN
1870         SELECT l.language, SUM(bc.amount * b.price)
1871         FROM bookcart bc
1872         JOIN books b ON bc.bookid = b.id
1873         JOIN languages l ON b.languageid = l.id
1874         GROUP BY l.language
1875     LOOP
1876         RAISE NOTICE 'Language: %           Total Sales: %', language, total_sales;
1877     END LOOP;
1878 END;
1879 $$;
1880
1881 CALL calculate_sales_by_language();
```

Data Output | Messages | Notifications

```
NOTICE:  Sales by language:
NOTICE:  Language: English          Total Sales: 170100.00
NOTICE:  Language: Sinhala          Total Sales: 183900.00
NOTICE:  Language: Japanese          Total Sales: 188200.00
NOTICE:  Language: Tamil          Total Sales: 143600.00
```

Figure 2.6.3: sales in languages

The calculate_sales_by_language procedure calculates the total sales for each book language in the system. It joins the bookcart, books, and languages tables to compute the sales amount by multiplying the number of books sold with their respective prices. The results are grouped by language, and the procedure displays each language along with its total sales using notices. It provides an overview of sales distribution across different languages. (see Figure 2.6.3)

2.7. Transactions

A transaction in PostgreSQL is a sequence of operations performed as a single logical unit of work. It ensures the integrity of the database by the ACID properties Transactions in PostgreSQL are initiated with the BEGIN keyword and completed with either a COMMIT (transaction successfully ended) or ROLLBACK (transaction unsuccessfully ended).

```
1669  CREATE OR REPLACE PROCEDURE paid_transaction(p_customerid INT)
1670  LANGUAGE plpgsql
1671  AS $$
1672  DECLARE total_amount NUMERIC(10, 2);
1673  BEGIN
1674      BEGIN;
1675          SELECT totalamount
1676          INTO total_amount
1677          FROM payment
1678          WHERE customerid = p_customerid;
1679          IF NOT FOUND THEN
1680              RAISE NOTICE 'No payment record found for customer ID %.', p_customerid;
1681              RETURN;
1682          END IF;
1683          RAISE NOTICE 'Total amount for customer ID %: %', p_customerid, total_amount;
1684
1685          INSERT INTO paid (customerid, totalprice, paid_date)
1686          VALUES (p_customerid, total_amount, NOW());
1687
1688          DELETE FROM payment WHERE customerid = p_customerid;
1689          DELETE FROM bookcart WHERE customerid = p_customerid;
1690
1691          RAISE NOTICE 'Transaction completed successfully for customer ID %.', p_customerid;
1692
1693      EXCEPTION WHEN OTHERS THEN
1694          RAISE NOTICE 'Transaction failed.';
1695          ROLLBACK;
1696      END;
1697  END;
1698  $$;
1699
1700  CALL paid_transaction(100);
```

Figure 2.7.1: transaction on payment

The paid_transaction procedure handles a customer's payment process as a single transaction in PostgreSQL. It retrieves the total amount for the given customer from the payment table, records the payment in the paid table with the current date, and then removes the customer's entries from both the payment and bookcart tables. Using explicit transaction control ensures all operations succeed together or are rolled back if an error occurs, maintaining database consistency.

```
1884  DO $$
1885  BEGIN
1886      -- Check if the language exists in the table
1887      IF NOT EXISTS (SELECT 1 FROM languages WHERE language = 'English') THEN
1888
1889          INSERT INTO languages (language)
1890          VALUES ('English');
1891
1892          COMMIT;
1893          RAISE NOTICE 'Language inserted and transaction committed successfully.';
1894      ELSE
1895          ROLLBACK;
1896          RAISE NOTICE 'Transaction rolled back: Language already exists.';
1897      END IF;
1898  END;
1899  $$;
1900
1901
1902
```

Data Output | Messages | Notifications

```
NOTICE:  Transaction rolled back: Language already exists.
DO
```

Figure 2.7.2: Transaction on language

This transaction handles transactions with conditional logic. It checks if a language (e.g., "English") already exists in the languages table. If the language does not exist, it inserts the language and commits the transaction. If the language already exists, it rolls back the transaction, ensuring that no changes are made.

2.8. Summary

In this chapter explores key database management concepts, including indexing, queries, transactions, functions, triggers, and procedures. Indexing optimizes data retrieval by creating efficient lookup structures, while queries form the foundation for interacting with data through operations like filtering, joining, and aggregation. Transactions ensure data consistency and reliability by grouping multiple operations into a single, atomic unit. Functions and procedures encapsulate reusable logic, with functions focusing on returning results and procedures executing complex, multi-step tasks. Triggers automate actions in response to database events, maintaining data integrity and enforcing business rules. Together, these tools provide a robust framework for efficient, secure, and scalable database management.

Chapter 3 : CONCLUSION

In conclusion, the Book Heaven project aims to develop a robust and efficient database management system that facilitates seamless operations for an online bookstore. By leveraging PostgreSQL, the system is designed to handle key functions such as inventory management, customer transactions, and author data with high efficiency. The use of indexing, queries, transactions, functions, triggers, and procedures ensures optimal data handling and integrity, while also enabling automated processes for tasks like payment processing and book management. The project's successful implementation will enhance the overall user experience, streamline business operations, and provide a scalable solution for future growth in the digital book marketplace.

REFERENCES

www.postgresql.org

www.Chatgpt.com

www.youtube.com