Sami Mattila & Roope Korhonen 11.10.2024

School Database Project

Introduction

This project aims to design, implement, and optimize a database system incorporating various database management techniques and tools learned throughout the course. The project will evolve from basic database design to advanced concepts such as query optimization, locking mechanisms, ORM with JPA, and usage of triggers, stored procedures, and events. The final outcome will be a well-designed, optimized, and maintainable database for a school management system.

School Management System Database

Objective:

Develop a functional and efficient school management database using SQL, optimizing queries, managing concurrency with locks and versioning, and leveraging ORM with JPA for a Java-based front end.

Scope:

Entities: Student, Teacher, Course, Enrolment, Department, etc.

Functionalities:

Student and teacher management.

Course and department management.

Enrolment system to track student-course relationships.

Event scheduling (e.g., exams, lectures).

User role management (student, teacher, admin).

Tools & Technologies:

Database Management Tool: HeidiSQL

ORM Framework: JPA (Java Persistence API)

SQL Dialect: MariaDB

Programming Language: Java (for ORM integration)

Documentation of the project

By following this documentation, the reader can easily follow the steps in order to re-create the project and test out the functionality. Let’s begin.

Step 1: Create the Tables

```
-- Create the database for the School Management System
CREATE DATABASE SchoolManagementSystem;
USE SchoolManagementSystem;
```

The database SchoolManagementSystem is created, and the USE SchoolManagementSystem; command ensures that the subsequent table creation statements will be executed in that database.

```
-- Create Department Table (must be created first since it's referenced by other tables)
CREATE TABLE Department (
    department_id INT PRIMARY KEY AUTO_INCREMENT,
    department_name VARCHAR(100)
);
-- Create Student Table
CREATE TABLE Student (
    student_id INT PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    date_of_birth DATE,
    department_id INT,
    version INT DEFAULT 1,
    CONSTRAINT fk_department FOREIGN KEY (department_id) REFERENCES
Department(department_id)
);
```

```
-- Create Teacher Table
CREATE TABLE Teacher (
    teacher_id INT PRIMARY KEY AUTO_INCREMENT,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    department_id INT,
    CONSTRAINT fk_teacher_department FOREIGN KEY (department_id) REFERENCES Department(department_id)
);
-- Create Course Table
CREATE TABLE Course (
    course_id INT PRIMARY KEY AUTO_INCREMENT,
    course_name VARCHAR(100),
    teacher_id INT,
    department_id INT,
    CONSTRAINT fk_course_teacher FOREIGN KEY (teacher_id) REFERENCES Teacher(teacher_id),
    CONSTRAINT fk_course_department FOREIGN KEY (department_id) REFERENCES Department(department_id)
);
-- Create Enrollment Table (Many-to-Many between Student and Course)
CREATE TABLE Enrollment (
    enrollment_id INT PRIMARY KEY AUTO_INCREMENT,
    student_id INT,
    course_id INT,
    enrollment_date DATE,
    grade DECIMAL(3,2),
    CONSTRAINT fk_enrollment_student FOREIGN KEY (student_id) REFERENCES Student(student_id),
    CONSTRAINT fk_enrollment_course FOREIGN KEY (course_id) REFERENCES Course(course_id));
```

The Department table is created first because both the Student and Teacher tables reference it via foreign keys.

After that, the Student, Teacher, Course, and Enrollment tables are created, maintaining the proper foreign key relationships.

Step 2: Insert Sample Data

We need some sample data to work with for testing queries, triggers, and procedures. Let's insert data into the Department, Student, Teacher, Course, and Enrollment tables.

```
-- Insert into Department
INSERT INTO Department (department_name) VALUES ('Computer Science');
INSERT INTO Department (department_name) VALUES ('Mathematics');
INSERT INTO Department (department_name) VALUES ('Physics');

-- Insert into Student
INSERT INTO Student (first_name, last_name, date_of_birth, department_id)
VALUES ('John', 'Doe', '2001-05-15', 1);
INSERT INTO Student (first_name, last_name, date_of_birth, department_id)
VALUES ('Jane', 'Smith', '2000-11-22', 2);

-- Insert into Teacher
INSERT INTO Teacher (first_name, last_name, department_id)
VALUES ('Alice', 'Brown', 1);
INSERT INTO Teacher (first_name, last_name, department_id)
VALUES ('Bob', 'Green', 2);

-- Insert into Course
INSERT INTO Course (course_name, teacher_id, department_id)
VALUES ('Algorithms', 1, 1);
INSERT INTO Course (course_name, teacher_id, department_id)
VALUES ('Calculus', 2, 2);
```

```
-- Alter the table before inserting values into enrollment_date
ALTER TABLE Enrollment MODIFY grade DECIMAL(5,2);

-- Insert into Enrollment (Students enrolled in courses)

INSERT INTO Enrollment (student_id, course_id, enrollment_date, grade)
VALUES (1, 1, '2024-01-15', 95.5);
INSERT INTO Enrollment (student_id, course_id, enrollment_date, grade)
VALUES (2, 2, '2024-01-16', 89.0);
```

Step 3: Query Optimization

Now that we have some data, we can run and optimize some queries to retrieve information. We will be also explaining how to optimize these queries using indexes.

Query_1: Get All Students in a Specific Department

```
-- Basic Query: Get all students from the 'Computer Science' department
SELECT s.first_name, s.last_name, d.department_name
FROM Student s
JOIN Department d ON s.department_id = d.department_id
WHERE d.department_name = 'Computer Science';
```

Optimizing Query_1

We'll add an index on the department_name field of the Department table to optimize this query. This will make searching by department faster.

```
CREATE INDEX idx_department_name ON Department(department_name);
```

Query_2: Get All Courses a Student Is Enrolled In

```
SELECT s.first_name, s.last_name, c.course_name
FROM Student s
JOIN Enrollment e ON s.student_id = e.student_id
JOIN Course c ON e.course_id = c.course_id
WHERE s.first_name = 'John' AND s.last_name = 'Doe';
```

Optimizing Query_2

Adding an index on the student_id field of the Enrollment table and the course_id field will help speed up this query.

CREATE INDEX idx_student_id ON Enrollment(student_id);

CREATE INDEX idx_course_id ON Enrollment(course_id);

Step 3.1: Summary of what we have just done and why

These steps focus on improving the performance of SQL queries by introducing optimization techniques. Key methods used for query optimization include:

Indexing: Indexes are added to columns that are frequently used in JOIN, WHERE, and ORDER BY clauses to speed up data retrieval. For example, indexing the department_name column in the Department table helps to speed up searches by department.

Optimizing Joins: Queries that involve multiple tables (e.g., JOIN between Student, Enrollment, and Course tables) are optimized by adding indexes on the columns involved in these joins, like student_id and course_id in the Enrollment table.

Use of EXPLAIN: The EXPLAIN statement is used to analyze query execution plans and identify any inefficiencies. This helps in deciding where to add indexes and how to structure queries better.

By using these techniques, queries will run faster and more efficiently, especially as the data in the tables grows larger.

Step 4: Locks & Versioning

For concurrency control and ensuring data consistency when multiple users are trying to update the same record, we'll use optimistic locking with a version column and pessimistic locking with transactions.

Optimistic Locking

The version column is used to implement optimistic locking, which helps prevent conflicts during updates.

```
UPDATE Student
SET first_name = 'Johnny', version = version + 1
WHERE student_id = 1 AND version = 1;
```

If two users try to update the same student at the same time, the first update will succeed, and the second one will fail if the version number has already been changed.

Pessimistic Locking

We can use SELECT FOR UPDATE to lock a row while performing updates, ensuring no other transaction can modify it.

```
START TRANSACTION;
SELECT * FROM Student WHERE student_id = 1 FOR UPDATE;
UPDATE Student SET first_name = 'Johnathan' WHERE student_id = 1;
COMMIT;
```

Step 5: Triggers

In this step, we focus on implementing database triggers, which are essential components for automating tasks and managing database integrity in a relational database system like MariaDB.

Triggers

A trigger is a special type of stored procedure that automatically executes (or "fires") in response to certain events on a particular table, such as INSERT, UPDATE, or DELETE.

Purpose of Triggers

Maintain Data Integrity: Automatically enforce business rules without requiring explicit application logic.

Audit Changes: Track changes to data, like maintaining a history of changes or recording user actions.

Automate Actions: Automatically perform operations when specific changes occur, like updating related tables or calculating summary data.

Implementation: In this project, we created a simple trigger to automatically update the student_count in the Course table every time a new student enrolls in a course. Here’s a summary of how it was done:

Trigger Name: update_course_student_count

Trigger Timing: AFTER INSERT on the Enrollment table.

Action: The trigger increments the student_count for the corresponding course_id whenever a new row is added to the Enrollment table.

Code Example:

```
CREATE TRIGGER update_course_student_count
AFTER INSERT ON Enrollment
FOR EACH ROW
UPDATE Course
SET student_count = student_count + 1
WHERE course_id = NEW.course_id;
```

Testing:

After inserting a record into the Enrollment table, we checked the student_count in the Course table to verify that it updated correctly. The test confirmed that the trigger was functioning as intended.

Detailed steps how to test that the Trigger actually works. First, make sure that the Course table has a student_count column. If not, add it by doing the following code:

```
ALTER TABLE Course ADD student_count INT DEFAULT 0;
```

Next, insert a new row into the Enrollment table and check if the student_count in the corresponding Course is updated:

```
INSERT INTO Enrollment (student_id, course_id, enrollment_date, grade)
VALUES (1, 1, '2024-02-15', 87.0);
```

Lastly, check the COURSE table to see if the `student_count` in the `Course` table has increased by 1

```
SELECT course_name, student_count
FROM Course
```

```
WHERE course_id = 1;
```

Summary of step 5

In this step, we successfully implemented a simple trigger to maintain data integrity and automate updates within our database. The focus was on understanding the purpose and functionality of triggers, with future potential for stored procedures and events to further enhance the database's capabilities.

Step 6: Java and JPA Integration in Eclipse

In case you aren't familiar with Eclipse, here is a detailed run down with step-by-step instructions to setup the project. Hold on to your seat, this step is the longest one!

1. Set Up the Java Project with Maven

Open Eclipse.

Create a New Maven Project:

Click on File > New > Other…

In the wizard, expand the Maven folder and select Maven Project, then click Next.

Check the box for Create a simple project (skip archetype selection) and click Next.

Fill in Project Details:

Group Id: com.example

Artifact Id: student-course-app

Leave the other fields as default and click Finish.

Update Project Structure:

In the Package Explorer, right-click on your project (e.g., student-course-app).

Click on Properties, then select Java Build Path.

Click on the Libraries tab and ensure you have JDK set up properly. If not, add your JDK.

Add Maven Dependencies:

Open your pom.xml file (found in the root of your project).

Inside the <dependencies> tag, add the following dependencies:

```
<dependencies>
    <!-- JPA and Hibernate Dependencies -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.6.14.Final</version>
    </dependency>
    <dependency>
        <groupId>javax.persistence</groupId>
        <artifactId>javax.persistence-api</artifactId>
        <version>2.2</version>
    </dependency>
    <!-- MySQL Connector Dependency -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.33</version>
    </dependency>
    <!-- JUnit for Testing (optional) -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
```

```
<!-- MariaDB JDBC Driver -->
        <dependency>
            <groupId>org.mariadb.jdbc</groupId>
            <artifactId>mariadb-java-client</artifactId>
            <version>3.1.2</version>
        </dependency>
</dependencies>
```

Save the pom.xml. Eclipse will automatically download the required libraries.

Define JPA Entities

Next up we will create and define JPA Entities.

Class Names and Corresponding Database Tables:

Student (corresponds to the Student table)

Teacher (corresponds to the Teacher table)

Course (corresponds to the Course table)

Enrollment (corresponds to the Enrollment table)

Department (corresponds to the Department table)

Step-by-Step Instructions for Entity Creation

1. Create the Package for Entities

Right-click on src/main/java in the Package Explorer.

Select New > Package.

Name the package com.example.entities and click Finish.

2. Create Entity Classes

For each entity, follow the steps:

Right-click on the entities package > New > Class.

Name the class (e.g., Student) and click Finish.

Inside each class, add the required attributes and JPA annotations as follows:

Student Entity

```
package com.example.entities;

import javax.persistence.*;
import java.util.Date;

@Entity
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int student_id;

    private String first_name;
    private String last_name;
    private Date date_of_birth;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    @Version
    private int version;

    // Getters and Setters
}
```

Teacher Entity

```
package com.example.entities;

import javax.persistence.*;

@Entity
public class Teacher {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int teacher_id;

    private String first_name;
    private String last_name;
    private String email;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    // Getters and Setters
}
```

Course Entity

```
package com.example.entities;

import javax.persistence.*;

@Entity
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int course_id;

    private String course_name;
    private int student_count;

    @ManyToOne
    @JoinColumn(name = "teacher_id")
    private Teacher teacher;

    // Getters and Setters
}
```

Enrollment Entity

```
package com.example.entities;

import javax.persistence.*;
import java.util.Date;

@Entity
public class Enrollment {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int enrollment_id;

    @ManyToOne
    @JoinColumn(name = "student_id")
    private Student student;

    @ManyToOne
    @JoinColumn(name = "course_id")
    private Course course;

    private Date enrollment_date;
    private float grade;
```

```
    // Getters and Setters
}
```

Department Entity

```
package com.example.entities;

import javax.persistence.*;

@Entity
public class Department {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int department_id;

    private String department_name;

    // Getters and Setters
}
```

Step 3: Add Getters and Setters

For each class, we'll want to add getters and setters for the attributes. We can use Eclipse's built-in feature to generate them:

Open the class (e.g., Student.java).

Go to the menu bar and click on Source > Generate Getters and Setters....

Select all fields and click Generate.

Repeat this process for each class.

Now that our Java project has JPA entities that are corresponding to our database tables, we need to create a persistence.xml file.

Creating the persistence.xml File

The persistence.xml file is where you configure the JPA settings, database connection, and entities for your project.

Steps to Create persistence.xml in Eclipse:

Right-click on src/main/resources > New > Other.

In the dialog box, type persistence.xml in the search field.

Select Persistence under JPA and click Next.

In the next window, select the default options and click Finish.

If there isn’t META-INF folder created, you can simply create it inside resources and drag your persistence.xml there. After this is done, we will configure persistence.xml

Configuration for persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.2">
   <persistence-unit name="student-course-app" transaction-type="RESOURCE_LOCAL">

      <!-- JPA Provider -->
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

      <!-- Database Connection Settings -->
      <properties>
         <!-- JDBC Connection -->
         <property name="javax.persistence.jdbc.driver" value="org.mariadb.jdbc.Driver"/>
         <property name="javax.persistence.jdbc.url"
value="jdbc:mariadb://localhost:3306/your_database_name"/>
         <property name="javax.persistence.jdbc.user" value="your_username"/>
         <property name="javax.persistence.jdbc.password" value="your_password"/>

         <!-- Hibernate Properties -->
         <property name="hibernate.dialect"
value="org.hibernate.dialect.MariaDB103Dialect"/>
         <property name="hibernate.hbm2ddl.auto" value="update"/>
         <property name="hibernate.show_sql" value="true"/>
         <property name="hibernate.format_sql" value="true"/>
      </properties>
```

```
    </persistence-unit>
</persistence>
```

Remember to change the “your_database_name”, “your_username” and “your_password” with the real values in order to make the connection work.

Explanation of the Key Sections

persistence-unit

The name attribute (student-course-app) should be unique and will be used to reference the persistence context.

transaction-type="RESOURCE_LOCAL" means that transactions will be managed by the application (not a JTA provider).

provider

Specifies the JPA provider you are using. In this case, we use Hibernate (org.hibernate.jpa.HibernatePersistenceProvider).

Database Connection Settings

javax.persistence.jdbc.driver: Set to org.mariadb.jdbc.Driver for MariaDB.

javax.persistence.jdbc.url: The connection URL to your MariaDB database. Replace your_database_name with the actual name of your database.

javax.persistence.jdbc.user and javax.persistence.jdbc.password: Set your database username and password here.

Hibernate Properties

hibernate.dialect: This specifies the dialect for MariaDB. You should use MariaDB103Dialect (or later) for MariaDB.

hibernate.hbm2ddl.auto

update: Automatically creates/updates the tables based on your entity classes.

Other options are create, create-drop, validate, and none.

hibernate.show_sql and hibernate.format_sql: These properties display and format the SQL being generated by Hibernate.

Once all of the previous steps have been done, it's time to test the connection itself. If there occur any problems with the connection, you may assume it's because of Eclipse's correct setup. Be sure you have your JDK configured correctly. Anyways, moving on to the test.

Test Persistence Setup

Let's create a simple Main class to test our configuration. Create this Main class inside our src/main/java folder.

```
package com.example;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Main {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("student-course-app");
        EntityManager em = emf.createEntityManager();

        // If this runs without exceptions, it means the configuration is correct.
        System.out.println("JPA setup is working correctly!");

        em.close();
        emf.close();
```

```
    }
}
```

Run the Main Class

Right-click on Main.java in the Package Explorer.

Select Run As > Java Application.

If the setup is correct, the console in Eclipse should print:

“JPA setup is working correctly!”

This confirms that your JPA configuration (including persistence.xml) is working correctly and connected to your MariaDB database.

Here's a summary of what this project has accomplished

1. Database Design and Implementation

Structured Database: Created a well-defined schema for a school management system, including tables for Student, Teacher, Course, Department, and Enrollment.

Relationships: Implemented foreign key relationships that model real-world relationships (e.g., students enrolling in courses, teachers assigned to departments).

2. ORM with JPA

Entity Mapping: Mapped Java classes to database tables using JPA annotations, enabling seamless interaction between Java objects and database records.

Persistence Context: Managed the lifecycle of entities with the help of JPA, which simplifies database operations by handling the underlying SQL.

3. Concurrency Control

Optimistic and Pessimistic Locking: Implemented concurrency control mechanisms to ensure data consistency in multi-user environments, preventing data conflicts during simultaneous updates.

4. Triggers and Database Automation

Triggers: Created triggers to automate updates (e.g., updating student counts in courses upon enrollment), demonstrating an understanding of database automation and event-driven architecture.

5. Future Scalability

Prepared for Extensions: The architecture is set up to easily add additional features like handling relationships, complex queries using JPQL or Criteria API, stored procedures for business logic, and scheduled events for periodic tasks.

6. Testing and Validation

End-to-End Testing: Set up the framework to test the entire application, ensuring that all components work together effectively.

Data Integrity: Ensured that the application maintains data integrity through locking and versioning.

7. Understanding of Advanced Concepts

Database Optimization: Gained experience with query optimization techniques, indexing, and performance testing, which are crucial for developing efficient database applications.

Handling Associations: Prepared to handle various associations between entities, which are important for accurately modeling relationships in the application.

Conclusion

Overall, this project represents a comprehensive learning experience that covers the essential components of building a data-driven Java application. It showcases your ability to design and implement a database schema, perform data manipulation, and leverage JPA for ORM. By completing this project, you've developed skills that are highly relevant in modern software development, particularly in the context of enterprise applications.

The potential of this project

This school management system project could be utilized in various educational and administrative settings. Here are some potential applications and environments where this project can be effectively used:

1. Educational Institutions

Schools: Manage student enrollment, course assignments, teacher-student relationships, and departmental management. The system can help in tracking student performance and attendance.

Colleges and Universities: Handle complex relationships between students, courses, and instructors. The system can support academic records management, degree tracking, and course registration.

2. Online Learning Platforms

E-Learning Systems: Facilitate online course management, including enrollment in digital courses, tracking progress, and managing instructor assignments. The system can also help manage student feedback and course evaluations.

3. Administrative Applications

Administrative Offices: Streamline operations in educational institutions by providing administrative staff with tools to manage records, generate reports, and automate notifications (e.g., reminders for enrollment deadlines).

Human Resources: Manage staff records, including teacher information and department assignments, improving internal HR operations.

4. Educational Software Solutions

Custom Software Development: Serve as a foundational project for companies developing custom software solutions tailored to educational needs. It can be adapted or extended to meet specific client requirements.

Integrated Learning Management Systems (LMS): Integrate with existing LMS solutions to provide a comprehensive approach to student management and course administration.

5. Research and Development

Educational Research: Use the system as a dataset for educational research, enabling studies on student performance, course effectiveness, or resource allocation.

Data Analysis Projects: Serve as a practical example for data analysts looking to apply techniques in educational contexts, such as analyzing enrollment trends or student demographics.

6. Training and Skill Development

Training Programs: Act as a project for training software developers in database management, JPA, and full-stack application development.

Educational Workshops: Use it as a case study in workshops or seminars that focus on building Java applications with database integration.

7. Mobile and Web Applications

Frontend Development: Serve as the backend for a mobile or web application, providing a RESTful API for client applications that interact with student data.

API Development: Extend the system to expose APIs for third-party applications, allowing integration with other educational tools or platforms.

Conclusion

The school management system project can be highly beneficial in various educational contexts, administrative settings, and software development initiatives. It provides a solid foundation for building more complex systems and can be tailored to fit different requirements, making it versatile for numerous applications in the education sector and beyond.