HO CHI MINH CITY UNIVERSITY OF TRANSPORT

Kiến thức - Kỹ năng - Sáng tạo - Hội nhập

Sứ mệnh - Tầm nhìn

Triết lý Giáo dục - Giá trị cốt lõi

Contents

1 Relational Database Schema: Student Management (Basic).

1. **Subject (SubjectID, SubjectName, Unit)**
   Predicate: Each subject **(Subject)** is assigned a unique code **(SubjectID)** to distinguish it from other subjects. The subject name **(SubjectName)** and the number of unit **(Unit)** for that subject are known..

2. **Class (ClassID, ClassName, ClassYear)**
   Predicate: Each class **(Class)** is assigned a unique code **(ClassID)** to distinguish it from other classes. The class name **(ClassName)** and the class year **(ClassYear)** are known.

3. **Student (StudentID, StudentName, StudentAddress, ClassID)**
   Predicate: Every student **(Student)** has a unique code **(StudentID)** that differentiates them from other students. The student's name **(StudentName)**, address **(StudentAddress)**, and class **(ClassID)** are recorded.

4. **StudentGrade (StudentID, SubjectID, Grade)**
   Predicate: The Student grade relational schema **(StudentGrade)** records the grade **(Grade)** of students **(StudentID)** for subjects **(SubjectID)**.

Require

1. Find all keys of the Relation Schemas.

2. Create database **DB01**.

3. Create the Relation Schemas.

4. Insert data:

   - **Subject**. SubjectID: S01 → S05
   - **Class**. ClassID: C01 → C03
   - **Student**. StudentID: T01 → T20
   - **StudentGrade**. Distribute grade of subject to the students. There are one to three subjects for each student. Only one half students have grade.

5. **Performing Queries with Relational Algebra and SQL:**

   5.1. Find the students that belong to class ID "C02".
   5.2. Find the students that belong to class name = "Computer Science".
   5.3. Find the students (All information) whose class year is "2023"
   5.4. Find the Subject (Name and Unit) of the Subject ID "S01".
   5.5. What are the grade of student "T02" for subject "S02"?
   5.6. Find the subjects (ID, Name, and Grade) in which student "T02" failed.
   5.7. Which subjects (*) did student "T03" never take the exam for?
   5.8. How many students are in each class?
   5.9. Find the classes with the largest number of students.
   5.10. Calculate the GPA (Grade Point Average) of student ID "T02".
   5.11. Calculate the GPA for each student.
   5.12. Calculate the GPA of class ID "C02".
   5.13. Calculate the GPA for each class.
   5.14. Find the students who have the largest GPA.
   5.15. Find the students (ID and Name) who have the largest GPA.
   5.16. Find the classes that have the largest average GPA.
   5.17. Find the classes (ID and Name) that have the largest average GPA.
   5.18. Calculate the GPA with weights for each student.
   5.19. Weighted GPA calculation for each student (ID and name).
   5.20. Which students have received a grade in all subjects?
   5.21. Which students have received a grade in all subjects where each subject has 2 units?
   5.22. Which students have passed (Grade >= 5) in all subjects where each subject has 2 units?

6. **Integrity Constraints:**

6.1. Add a NOT NULL constraint to ensure subject name is always provided.
6.2. Add a UNIQUE constraint to prevent duplicate subject names.
6.3. Add a CHECK constraint to ensure that Unit > 0.
6.4. Add a NOT NULL constraint to ensure class name and class year are always provided.
6.5. Add a UNIQUE constraint to prevent duplicate (class name, class year).
6.6. Add a NOT NULL constraint to ensure student name is always provided.
6.7. Add a FOREIGN KEY constraint to link Student.ClassID to Class(ClassID).
6.8. Add a CHECK constraint to ensure that grade are between 0 and 10.
6.9. Add a FOREIGN KEY constraint to link Studentgrade.SyudentID to Student(StudentID).
6.10. Add a FOREIGN KEY constraint to link Studentgrade.SubjectID to Subject(SubjectID).

7. **Store Procedure:**

7.1. Create a function to calculate the average grade of a student.
7.2. Create a function to get the class name from a student ID.
7.3. Create a procedure to insert a new student into the database.
7.4. Create a procedure to update a student's grade for a subject.
7.5. Create a function to classify student performance based on average grade.
7.6. Create a function to list subjects that a student has not taken yet.
7.7. Create a function to list students with an average grade below 5.0.
7.8. Create a procedure to delete a student and all their related grades.
7.9. Create a function to calculate the total number of units a student has earned.
7.10. Create a procedure to assign a new subject to all students with default grade.

8. **Index, View:**

8.1. Create an index on the StudentName column in the Student relation.
8.2. Create a view View_StudentInfo that shows student names and their class names.
8.3. Create a composite index on StudentGrade(StudentID, SubjectID).
8.4. Create a view View_StudentGrades showing student name, subject name, and grade.
8.5. Create an index on Subject(Unit) to improve performance of queries by unit count.
8.6. Create a view View_HighGrades that shows students with grade >= 8.
8.7. Create a partial index on StudentGrade(Grade) where Grade >= 8.
8.8. Create a materialized view MatView_AvgGradePerClass that stores the average grade per class.
8.9. Create a view View_FailedSubjects that shows students who failed at least one subject (grade < 5).
8.10. Create a unique index on Subject(SubjectName) to ensure no two subjects have the same name.

9. **Query Processing (Physical - Database Design):**

9.1. Compare the performance of queries with and without indexes:

```
SELECT *
FROM Student
WHERE ClassID = 'C101';
```

- Predict whether PostgreSQL will use a **Sequential Scan** or an **Index Scan**.
- Create an index on ClassID:

  CREATE INDEX idx_student_classid ON Student(ClassID);
- Rerun the query with Explain Analyze and compare performance.
- Disable index scan:

  SET enable_indexscan = OFF;
- Observe and explain the difference in the query plan and execution time.

9.2. Explore join strategies (Hash Join, Nested Loop, Merge Join)

```
SELECT StudentName, SubjectName, Grade
FROM Student A JOIN StudentGrade B ON A.StudentID = B.StudentID
     JOIN Subject C ON B.SubjectID = C.SubjectID
WHERE Grade > 80;
```

- Predict which join strategy PostgreSQL will use.
- Create indexes to optimize joins:

  ```
  CREATE INDEX idx_grade_studentid ON StudentGrade(StudentID);
  CREATE INDEX idx_grade_subjectid ON StudentGrade(SubjectID);
  ```
- Use EXPLAIN ANALYZE to observe the plan.
- Force a Nested Loop by disabling hash and merge joins:

  ```
  SET enable_hashjoin = OFF;
  SET enable_mergejoin = OFF;
  ```

9.3. Observe the effect of ORDER BY and filtering with indexes

```
SELECT s.StudentName, sg.Grade
FROM Student A JOIN StudentGrade B ON A.StudentID = B.StudentID
WHERE Grade BETWEEN 7.5 AND 8.5
ORDER BY Grade DESC;
```

- Use EXPLAIN ANALYZE to check whether PostgreSQL uses an index for sorting.
- Create a descending index:

  ```
  CREATE INDEX idx_grade_desc ON StudentGrade(Grade DESC);
  ```
- Rerun the query and compare performance with and without the index.

9.4. Compare performance of IN vs EXISTS

Version A (IN):

```
SELECT StudentName
FROM Student
WHERE StudentID IN ( SELECT StudentID
                     FROM StudentGrade
                     WHERE Grade >= 8.5 );
```

Version B (EXISTS):

```
SELECT StudentName
FROM Student A
WHERE EXISTS ( SELECT 1
               FROM StudentGrade
               WHERE A.StudentID = StudentID AND Grade >= 8.5 );
```

- Use EXPLAIN ANALYZE to observe and compare execution plans.
- Explain the differences in strategy and performance.
- Discuss which form is more efficient and why.

9.5. Analyze GROUP BY performance and the effect of indexing

```
SELECT ClassName, COUNT(StudentID) AS TotalStudents
FROM Class A JOIN Student B ON A.ClassID = B.ClassID
GROUP BY ClassName;
```

- Observe the execution plan using EXPLAIN ANALYZE.
- Create an index on Student(ClassID):

  ```
  CREATE INDEX idx_student_classid ON Student(ClassID);
  ```
- Rerun the query and compare the query plan and execution time.
- Explain how the index improves performance (or not).

10. Query Optimization (Physical - Database Design):

10.1. Selection Pushdown

```
SELECT StudentName
FROM Student A JOIN Class B ON A.ClassID = B.ClassID
WHERE ClassYear = 2023;
```

- Represent in relational algebra.
- Apply s**election pushdown** to the Class table.
- Rewrite the optimized SQL query.

10.2. Selection Pushdown on Both Tables

```
SELECT StudentName, ClassName
```

```
FROM Student A JOIN Class B ON A.ClassID = B.ClassID
WHERE StudentAddress LIKE '%Ho Chi Minh%' AND ClassYear = 2023;
```

- Push down both filters to Student and Class before the join.

10.3. Projection Pushdown

```
SELECT StudentName, Grade
FROM Student A JOIN StudentGrade B ON A.StudentID = B.StudentID
WHERE Grade > 8.5;
```

- Use projection pushdown to limit columns before the join.
- Explain the performance benefit.

10.4. Join Reordering

```
SELECT StudentName, SubjectName
FROM Student A JOIN StudentGrade B ON A.StudentID = B.StudentID
        JOIN Subject C ON B.SubjectID = C.SubjectID
WHERE SubjectName = 'Database Systems';
```

- Write the relational algebra expression.
- Apply selection pushdown to Subject.
- joins and rewrite optimized SQL.

10.5. Subquery to Join Conversion

```
SELECT StudentName
FROM Student
WHERE StudentID IN ( SELECT StudentID
                     FROM StudentGrade
                     WHERE Grade >= 8.5 );
```

- Rewrite using JOIN.
- Apply relational algebra optimizations.
- Compare performance before and after.

10.6. Multi-table Join Optimization

```
SELECT StudentName, SubjectName, Grade
FROM Student A JOIN StudentGrade B ON A.StudentID = B.StudentID
        JOIN Subject C ON B.SubjectID = C.SubjectID
WHERE Grade BETWEEN 7.0 AND 8.5;
```

- Predict join order.
- Push down filter to StudentGrade.
- Create appropriate indexes and analyze plans.

10.7. Pagination Optimization

```
SELECT * FROM Student ORDER BY StudentName LIMIT 50 OFFSET 5000;
```

- Create an index on Student(StudentName).
- Use EXPLAIN ANALYZE to analyze performance.
- Rewrite query using keyset pagination.

10.8. COUNT + Filter Optimization

```
SELECT COUNT(*) FROM StudentGrade WHERE Grade >= 8.5;
```

- Is an index helpful?
- Create an index on Grade.
- Compare plan and runtime with and without index.

10.9. CTE vs Subquery

CTE Version:

```
WITH TopStudents AS (
    SELECT StudentID
    FROM StudentGrade
    WHERE Grade >= 8.5 )

SELECT StudentName
FROM Student
```

```
WHERE StudentID IN ( SELECT StudentID FROM TopStudents);
```

Subquery Version:

```
SELECT StudentName
FROM Student
WHERE StudentID IN ( SELECT StudentID
                     FROM StudentGrade
                     WHERE Grade >= 8.5 );
```

- Compare query plans.
- Discuss whether the CTE is optimized or materialized.
- Explain pros/cons of each.

10.10. Deep Join Optimization

```
SELECT StudentName, ClassName, SubjectName, Grade
FROM Student A
    JOIN Class B ON A.ClassID = B.ClassID
    JOIN StudentGrade C ON B.StudentID = C.StudentID
    JOIN Subject D ON C.SubjectID = D.SubjectID
WHERE Grade > 8.5 AND ClassYear >= 2022 AND Unit = 3;
```

- Represent in relational algebra.
- Push selections down into Class, StudentGrade, Subject.
- Reorder joins by expected row size.
- Rewrite the optimized query and compare execution plan.

10.11. Aggregation + Join Optimization

```
SELECT ClassName, AVG(Grade) AS GPA
FROM Student A
    JOIN Class B ON A.ClassID = B.ClassID
    JOIN StudentGrade C ON B.StudentID = C.StudentID
GROUP BY ClassName
HAVING AVG(Grade) > 8.5;
```

- Push projections before joins.
- Estimate cost of aggregation.
- Optimize joins and compare results with EXPLAIN ANALYZE.

10.12. Subquery Rewrite with Join

```
SELECT StudentName
FROM Student
WHERE StudentID IN ( SELECT StudentID
                     FROM StudentGrade A JOIN Subject B ON A.SubjectID = B.SubjectID
                     WHERE SubjectName LIKE 'AI%' AND Grade > 8.5 );
```

- Convert to a join-based query.
- Optimize using selection pushdown and index usage.
- Show relational algebra before and after.

10.13. GROUP BY Subquery Optimization

```
SELECT StudentName
FROM Student
WHERE StudentID IN ( SELECT StudentID
                     FROM StudentGrade
                     GROUP BY StudentID
                     HAVING COUNT(DISTINCT SubjectID) > 10 );
```

- Create index on (StudentID, SubjectID).
- Estimate cost and plan.
- Rewrite or materialize if beneficial.

10.14. Optimizing View-Based Queries

Assume this view:

```
CREATE VIEW TopGrades AS
SELECT StudentID, Grade
FROM StudentGrade
```

WHERE Grade >= 8.5;

Query using the view:

```
SELECT StudentName
FROM Student A JOIN TopGrades B ON A.StudentID = B.StudentID
WHERE Grade < 10;
```

- Inline the view into the query.
- Combine conditions into a single selection: Grade BETWEEN 8.5 AND 10.
- Compare query plan with materialized view vs inlined logic.

2 Relational Database Schema: Student Management (Advance).

Create Database **DB02**

```
CREATE TABLE Department (
    DepartmentID TEXT PRIMARY KEY,
    DepartmentName TEXT NOT NULL
);
CREATE TABLE Class (
    ClassID TEXT PRIMARY KEY,
    ClassName TEXT NOT NULL,
    ClassYear INT NOT NULL,
    DepartmentID TEXT,
    FOREIGN KEY (DepartmentID) REFERENCES Department(DepartmentID)
);
CREATE TABLE Student (
    StudentID TEXT PRIMARY KEY,
    FullName TEXT NOT NULL,
    Gender CHAR(1),
    DateOfBirth DATE,
    Email TEXT,
    Phone TEXT,
    Address TEXT,
    ClassID TEXT,
    FOREIGN KEY (ClassID) REFERENCES Class(ClassID)
);
CREATE TABLE Subject (
    SubjectID TEXT PRIMARY KEY,
    SubjectName TEXT NOT NULL,
    Credits INT CHECK (Credits > 0),
    SubjectType TEXT CHECK (SubjectType IN ('Core', 'Elective'))
);
CREATE TABLE Semester (
    SemesterID TEXT PRIMARY KEY,
    Year INT,
    Term TEXT CHECK (Term IN ('Spring', 'Summer', 'Fall'))
);
CREATE TABLE Lecturer (
    LecturerID TEXT PRIMARY KEY,
    FullName TEXT,
    Email TEXT
);
CREATE TABLE Teaching (
    TeachingID SERIAL PRIMARY KEY,
    LecturerID TEXT ,
    SubjectID TEXT,
    SemesterID TEXT,
    FOREIGN KEY (LecturerID) REFERENCES Lecturer(LecturerID),
    FOREIGN KEY (SubjectID) REFERENCES Subject(SubjectID),
    FOREIGN KEY (SemesterID) REFERENCES Semester(SemesterID)
);
CREATE TABLE Enroll (
    EnrollID SERIAL PRIMARY KEY,
    StudentID TEXT,
    SubjectID TEXT ,
    SemesterID TEXT,
    Grade NUMERIC(3,1),
    FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
    FOREIGN KEY (SubjectID) REFERENCES Subject(SubjectID),
    FOREIGN KEY (SemesterID) REFERENCES Semester(SemesterID)
);
```

Generate Sample Data:

- **Department** – 3 departments
- **Class**: 20 classes
- **Subject**: 30 subjects
- **Semester**: 8 semesters (Spring & Fall từ 2021–2024)
- **Lecturer**: 10 lecturers
- **Student**: 1000 students
- **Teaching**: Subjects taught by lecturers on a semester basis
- **Enroll**: Students take 4–6 subjects per term, with a grade (0.0–10.0, step 0.5)

– 1. **Department**

```
INSERT INTO Department (DepartmentID, DepartmentName) VALUES
    ('DPT01', 'Computer Science'),
    ('DPT02', 'Mathematics'),
    ('DPT03', 'Physics');
```

– 2. **Class**

```
INSERT INTO Class (ClassID, ClassName, ClassYear, DepartmentID)
SELECT
    'C' || LPAD(i::text, 3, '0'),
    'Class ' || i,
    2020 + (i % 4),
    CASE WHEN i % 3 = 0 THEN 'DPT01'
        WHEN i % 3 = 1 THEN 'DPT02'
        ELSE 'DPT03' END
FROM generate_series(1, 20) i;
```

– 3. **Subject**

```
INSERT INTO Subject (SubjectID, SubjectName, Credits, SubjectType)
SELECT
    'SUB' || LPAD(i::text, 2, '0'),
    'Subject ' || i,
    (random() * 2 + 2)::int,
    CASE WHEN random() > 0.5 THEN 'Core' ELSE 'Elective' END
FROM generate_series(1, 30) i;
```

– 4. **Semester**

```
INSERT INTO Semester (SemesterID, Year, Term)
SELECT
    'S' || y || t, y, t
FROM unnest(ARRAY['Spring', 'Fall']) t,
    generate_series(2021, 2024) y;
```

– 5. **Lecturer**

```
INSERT INTO Lecturer (LecturerID, FullName, Email)
SELECT
    'L' || LPAD(i::text, 3, '0'),
    'Lecturer ' || i,
    'lecturer' || i || '@binhbat.ai'
FROM generate_series(1, 10) i;
```

– 6. **Student**

```
INSERT INTO Student (StudentID, FullName, Gender, DateOfBirth, Email, Phone, Address, ClassID)
SELECT
    'S' || LPAD(i::text, 4, '0'),
    'Student ' || i,
    CASE WHEN random() > 0.5 THEN 'M' ELSE 'F' END,
    date '2000-01-01' + (random() * 2000)::int,
    'student' || i || '@binhbat.ai',
    '012345' || (1000 + i)::text,
    'Address ' || i,
```

```
    'C' || LPAD((1 + mod(i, 20))::text, 3, '0')
FROM generate_series(1, 1000) i;
```

– 7. **Teaching**

```
INSERT INTO Teaching (LecturerID, SubjectID, SemesterID)
SELECT
    'L' || LPAD((1 + mod(i, 10))::text, 3, '0'),
    'SUB' || LPAD((1 + mod(i, 30))::text, 2, '0'),
    'S' || y || t
    FROM generate_series(1, 100) i,
        unnest(ARRAY['Spring', 'Fall']) t,
        generate_series(2021, 2024) y;
```

– 8. **Enroll**: SV học 4–6 môn mỗi kỳ, điểm hệ 10 (bước 0.5)

```
DO $$
DECLARE
    stu RECORD;
    sem RECORD;
    sub TEXT;
    sub_list TEXT[];
BEGIN
    FOR stu IN SELECT StudentID FROM Student LOOP
        FOR sem IN SELECT SemesterID FROM Semester LOOP
          sub_list := ARRAY(
          SELECT SubjectID FROM Subject ORDER BY random() LIMIT (4 + floor(random() * 3))::int
          );
          FOREACH sub IN ARRAY sub_list LOOP
            INSERT INTO Enroll(StudentID, SubjectID, SemesterID, Grade)
            VALUES (stu.StudentID, sub, sem.SemesterID, round(random() * 20) / 2.0);
          END LOOP;
        END LOOP;
    END LOOP;
END $$;
```

1. **Performing Queries with SQL:**

    1.1. Find students with the highest GPA each semester.

    1.2. Find the subject with the fewest students passed (Grase >= 5).

    1.3. Find students who fail at least 2 times in the same subject in 2 different periods.

    1.4. Identify Students Taking All Subjects.

    1.5. For each subject, the number of students is counted according to the grade levels (e.g., 0-2, 2-4, 4-6, 6-8, 8-10).

    1.6. Take a list of students who have taken the number of subjects they have taken and their GPA is higher than the school average, and sort them in descending order according to their GPA.

    1.7. Calculate the GPA of each class and compare it to the school"s overall GPA.

    1.8. Which class all the students have studied all the subjects.

    1.9. Which class all students have passed all the exams.

3 Relational Database Schema: Order Management.

1. **Category (CategoryID, CategoryName)**
   Predicate: Each category **(Category)** is assigned a unique code **(CategoryID)** to distinguish it from other categories. The category name **(CategoryName)** is known.
2. **Product (ProductID, ProductName, UnitPrice, CategoryID)**
   Predicate: Every product **(Product)** has a unique code **(ProductID)** that differentiates it from other products. The product name **(ProductName)**, unit price **(UnitPrice)**, and category **(CategoryID)** are recorded.
3. **Customer (CustomerID, CustomerName, CustomerAddress)**
   Predicate: Each customer **(Customer)** is assigned a unique code **(CustomerID)** to distinguish them from other customers. The customer's name **(CustomerName)** and address **(CustomerAddress)** are known.
4. **Orders (OrdersID, OrdersDate, RequiredDate, CustomerID)**
   Predicate:Every order **(Orders)** has a unique code **(OrdersID)** that differentiates it from other orders. The order date **(OrdersDate)**, required date **(RequiredDate)**, and the customer **(CustomerID)** who placed the order are recorded.
5. **OrdersDetail (OrdersID, ProductID, Quantity)**
   Predicate:The Order Detail relational schema **(OrdersDetail)** stores the quantity **(Quantity)** of each product **(ProductID)** in an order **(OrdersID)**.
6. **Delivery (DeliveryID, DeliveryDate, OrderID)**
   Predicate:Each delivery **(Delivery)** is assigned a unique code **(DeliveryID)** to distinguish it from other deliveries. The delivery date **(DeliveryDate)** and the order **(OrderID)** it fulfills are known.
7. **DeliveryDetail (DeliveryID, ProductID, Quantity)**
   Predicate: The Delivery Detail relational schema **(DeliveryDetail)** stores the quantity **(Quantity)** of each product **(ProductID)** in a delivery **(DeliveryID)**.

1. Find all keys of the Relation Schemas.
2. Create database **DB03**.
3. Create the Relation Schemas.
4. Insert all the required data for queries and integrity constraints.
5. **Performing Queries with Relational Algebra and SQL:**
   - 5.1. All products in category ID 'C02'
   - 5.2. List of customers who placed orders between d1 and d2.
   - 5.3. List of customers (ID, name, address) who placed orders in year 2025.
   - 5.4. List of products (ID) ordered in order ID 'D01'.
   - 5.5. List all product for order 'D01'.
   - 5.6. List all product (*) for orders on date 'd'.
   - 5.7. Calculating total quantities for each order (ID).
   - 5.8. Calculating the total quantity for each order (ID) in year 2025.
   - 5.9. Identify orders (ID) with the largest total Cost.
   - 5.10. In 2025, orders (ID) that had the highest total Cost.
   - 5.11. Computing the total Cost of orders for each customer.
   - 5.12. Identify customers (ID) with the largest total Cost.
   - 5.13. Calculating the total quantity for each customer (ID, name).
   - 5.14. In 2025, the total Cost of orders was calculated for each customer (ID, name).
   - 5.15. In 2025, customers (ID, name, address) with the highest total Cost of orders.
   - 5.16. Which Orders its place all products of category 'C01'
   - 5.17. Which Delivery did it deliver all the product of orders
   - 5.18. Create procedure to calculate Cost for each Order: OrdersCost()
   - 5.19. Create procedure to calculate Total Cost for each Customer: CustomerCost()
   - 5.20. Create a function to calculate Cost for Order: FOrderCost(OrderID)
   - 5.21. Create a function to calculate Total Cost for Customer: FCustomerCost(CustomerID)
6. **Show and Create all the integrity constraints:**

4 Relational Database Schema: Logistics Order Management

Crate Schema: **DB04**

```
CREATE TABLE Category (
    CategoryID SERIAL PRIMARY KEY,
    CategoryName VARCHAR(100) NOT NULL
);
CREATE TABLE Product (
    ProductID SERIAL PRIMARY KEY,
    ProductName VARCHAR(100) NOT NULL,
    UnitPrice NUMERIC(10,2) NOT NULL,
    CategoryID INTEGER NOT NULL,
    FOREIGN KEY (CategoryID) REFERENCES Category(CategoryID)
);
CREATE TABLE Customer (
    CustomerID SERIAL PRIMARY KEY,
    CustomerName VARCHAR(100) NOT NULL,
    CustomerAddress TEXT NOT NULL
);
CREATE TABLE Orders (
    OrderID SERIAL PRIMARY KEY,
    OrderDate DATE NOT NULL,
    RequiredDate DATE NOT NULL,
    CustomerID INTEGER NOT NULL,
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)
);
CREATE TABLE OrdersDetail (
    OrderID INTEGER ,
    ProductID INTEGER ,
    Quantity INTEGER CHECK (Quantity > 0),
    PRIMARY KEY (OrderID, ProductID),
    FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID)
);
CREATE TABLE Warehouse (
    WarehouseID SERIAL PRIMARY KEY,
    WarehouseName VARCHAR(100) NOT NULL,
    Location TEXT
);
CREATE TABLE Inventory (
    ProductID INTEGER ,
    WarehouseID INTEGER ,
    QuantityInStock INTEGER CHECK (QuantityInStock >= 0),
    PRIMARY KEY (ProductID, WarehouseID),
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID),
    FOREIGN KEY (WarehouseID) REFERENCES Warehouse(WarehouseID)
);
CREATE TABLE Shipper (
    ShipperID SERIAL PRIMARY KEY,
    ShipperName VARCHAR(100) NOT NULL,
    Phone VARCHAR(20)
);
CREATE TABLE Delivery (
    DeliveryID SERIAL PRIMARY KEY,
    DeliveryDate DATE NOT NULL,
    OrderID INTEGER ,
    ShipperID INTEGER ,
    FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
    FOREIGN KEY (ShipperID) REFERENCES Shipper(ShipperID)
);
```

```sql
CREATE TABLE DeliveryDetail (
    DeliveryID INTEGER,
    ProductID INTEGER,
    Quantity INTEGER CHECK (Quantity > 0),
    PRIMARY KEY (DeliveryID, ProductID),
    FOREIGN KEY (DeliveryID) REFERENCES Delivery(DeliveryID),
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID)
);
CREATE TABLE OrderStatus (
    OrderID INTEGER ,
    Status VARCHAR(30),
    UpdatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (OrderID) REFERENCES Orders(OrderID)
);
CREATE TABLE DeliveryStatus (
    DeliveryID INTEGER ,
    Status VARCHAR(30),
    UpdatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (DeliveryID) REFERENCES Delivery(DeliveryID)
);
CREATE TABLE OrderAddress (
    OrderID INTEGER PRIMARY KEY REFERENCES Orders(OrderID),
    AddressLine TEXT NOT NULL,
    City VARCHAR(50),
    PostalCode VARCHAR(20),
    Region VARCHAR(50),
    Country VARCHAR(50)
);
CREATE TABLE Invoice (
    InvoiceID SERIAL PRIMARY KEY,
    OrderID INTEGER ,
    InvoiceDate DATE NOT NULL,
    TotalAmount NUMERIC(12,2) NOT NULL,
    FOREIGN KEY (OrderID) REFERENCES Orders(OrderID)
);
CREATE TABLE Payment (
    PaymentID SERIAL PRIMARY KEY,
    InvoiceID INTEGER ,
    Amount NUMERIC(12,2) NOT NULL,
    PaymentDate DATE,
    Method VARCHAR(30),
    FOREIGN KEY (InvoiceID) REFERENCES Invoice(InvoiceID)
);
CREATE TABLE ShippingFee (
    OrderID INTEGER PRIMARY KEY REFERENCES Orders(OrderID),
    FeeAmount NUMERIC(10,2) NOT NULL
);
CREATE TABLE DeliveryTracking (
    TrackingID SERIAL PRIMARY KEY,
    DeliveryID INTEGER ,
    Timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    Location TEXT,
    StatusNote TEXT,
    FOREIGN KEY (DeliveryID) REFERENCES Delivery(DeliveryID) );
```

Generate Sample Data:

```sql
– 1. Category (10)
INSERT INTO Category(CategoryName)
SELECT ’Category ’ || i
FROM generate_series(1, 10) AS s(i);
```

```sql
– 2. Product (100)
```

```
INSERT INTO Product(ProductName, UnitPrice, CategoryID)
SELECT
    'Product ' || i,
    ROUND((random() * 490 + 10)::numeric, 2), – Price: 10 - 500
    (random() * 9 + 1)::int – CategoryID: 1 - 10
FROM generate_series(1, 100) AS s(i);

– 3. Customer (500)
INSERT INTO Customer(CustomerName, CustomerAddress)
SELECT
    'Customer ' || i,
    'Address ' || i
FROM generate_series(1, 500) AS s(i);

– 4. Warehouse (5)
INSERT INTO Warehouse(WarehouseName, Location)
SELECT
    'Warehouse ' || i,        'Location ' || i FROM generate_series(1, 5) AS s(i);

– 5. Inventory
INSERT INTO Inventory(ProductID, WarehouseID, QuantityInStock)
SELECT
    p, w, (random() * 1000)::int
FROM generate_series(1, 100) AS p,
       generate_series(1, 5) AS w;

– 6. Shipper (20)
INSERT INTO Shipper(ShipperName, Phone)
SELECT
    'Shipper ' || i,
    '0900' || lpad(i::text, 4, '0')
FROM generate_series(1, 20) AS s(i);

– 7. Orders + OrderDetails + ShippingFee + Invoice + Payment + Delivery
DO $$
DECLARE
    i INT;
    cust_id INT;
    order_dt DATE;
    required_dt DATE;
    delivery_dt DATE;
    shipper_id INT;
    fee NUMERIC;
    total NUMERIC;
    prod_id INT;
    qty INT;
BEGIN
    FOR i IN 1..1000 LOOP
        cust_id := (random() * 499 + 1)::int;
        order_dt := DATE '2023-01-01' + (random() * 700)::int;
        required_dt := order_dt + ((random() * 7)::int);
        delivery_dt := required_dt + ((random() * 3)::int);
        shipper_id := (random() * 19 + 1)::int;
        fee := ROUND((random() * 45 + 5)::numeric, 2);
        total := ROUND((random() * 1900 + 100)::numeric, 2);

        – Đơn hàng
        INSERT INTO Orders(OrderID, OrderDate, RequiredDate, CustomerID)
        VALUES (i, order_dt, required_dt, cust_id);

        – Chi tiết đơn hàng (1 đến 5 sản phẩm cho mỗi đơn)
        FOR prod_id IN (
            SELECT ProductID FROM Product ORDER BY random() LIMIT (1 + (random() * 4)::int)
        ) LOOP
            qty := (random() * 20 + 1)::int;
            INSERT INTO OrdersDetail(OrderID, ProductID, Quantity)
            VALUES (i, prod_id, qty);
        END LOOP;
```

```
    – Phí vận chuyển
    INSERT INTO ShippingFee(OrderID, FeeAmount)
    VALUES (i, fee);

    – Hóa đơn
    INSERT INTO Invoice(OrderID, InvoiceDate, TotalAmount)
    VALUES (i, delivery_dt, total);

    – Thanh toán
    INSERT INTO Payment(InvoiceID, Amount, PaymentDate, Method)
    VALUES (
        i,
        total,
        delivery_dt + 1,
        (ARRAY['Cash', 'Card', 'Transfer'])[floor(random() * 3 + 1)::int]
    );

    – Giao hàng
    INSERT INTO Delivery(DeliveryDate, OrderID, ShipperID)
    VALUES (delivery_dt, i, shipper_id);
  END LOOP;
END $$;
```

1. **Performing Queries with SQL**

   1.1. Find the top 5 customers with the highest total spending.

   1.2. List all orders that were delivered late compared to the RequiredDate.

   1.3. Find products that have never been ordered.

   1.4. For each customer, count the number of orders and calculate the total amount paid.

   1.5. Identify the top 3 best-selling products (based on total quantity sold).

   1.6. Find the shipper who has delivered the most orders.

   1.7. List the order that contains the highest number of products (in total quantity).

   1.8. Calculate the average inventory quantity for each product category.

   1.9. Find orders where the total shipping fee is greater than 10

   1.10. Use a window function to rank customers based on their total spending.

2. **Query Processing (Physical - Database Design)**

   2.1. Analyze the Join Strategy
   Execute the following query and use **EXPLAIN ANALYZE** to determine whether PostgreSQL uses a **Hash Join**, **Merge Join**, or **Nested Loop Join**:

```
EXPLAIN ANALYZE
SELECT c.CustomerName, p.ProductName, od.Quantity
FROM Orders o
JOIN Customer c ON o.CustomerID = c.CustomerID
JOIN OrdersDetail od ON o.OrderID = od.OrderID
JOIN Product p ON od.ProductID = p.ProductID;
```

   Instructions:

   - Identify the join strategy used.
   - Suggest how to influence PostgreSQL to use a different join strategy (e.g., by adding indexes or rewriting the query).

   2.2. Compare Query Performance With and Without Index
   Measure the performance of the following query before and after adding an index on the UnitPrice column:

```
– Query
EXPLAIN ANALYZE
SELECT * FROM Product WHERE UnitPrice > 300;

– Then add index
CREATE INDEX idx_unitprice ON Product(UnitPrice);
```

   Instructions:

   - Compare execution plans: Seq Scan vs Index Scan.
   - Analyze whether the index improves performance and under what conditions.

2.3. Rewrite Query Using Relational Algebra and Optimize It

```
SELECT * FROM Orders o
JOIN OrdersDetail od ON o.OrderID = od.OrderID
WHERE od.Quantity > 10 AND o.RequiredDate < CURRENT_DATE;
```

- First write the RA:

  $\sigma_{(od.Quantity>10 \land o.RequiredDate<today)}(Orders \bowtie OrdersDetail)$

- Then push selections to reduce intermediate result size:

  $\sigma_{(o.RequiredDate<today)}(Orders) \bowtie \sigma_{(od.Quantity>10)}(OrdersDetail)$

2.4. Predict Which Query Is Faster
Given the two queries below, which one will be faster? Why?

– Query A: using JOIN

```
SELECT o.OrderID
FROM Orders o
JOIN Customer c ON o.CustomerID = c.CustomerID
WHERE c.CustomerName ILIKE '%Smith%';
```

– Query B: using subquery

```
SELECT OrderID
FROM Orders
WHERE CustomerID IN (SELECT CustomerID
                     FROM Customer
                     WHERE CustomerName ILIKE '%Smith%') ;
```

Instructions:

- Use EXPLAIN to analyze both.
- Discuss cost differences between JOIN and IN.
- Evaluate advantages and drawbacks of each approach.

2.5. Optimize Monthly Revenue Aggregation
The query below computes monthly revenue from invoices. Suggest ways to optimize it:

```
SELECT DATE_TRUNC('month', InvoiceDate) AS Month, SUM(TotalAmount) AS Revenue
FROM Invoice
GROUP BY Month
ORDER BY Month;
```

Instructions:

- Use EXPLAIN ANALYZE to check scan type (sequential/index).
- Propose adding index on InvoiceDate.
- Consider denormalization (precomputing month) for frequent reporting.

3. Query Optimization (Physical - Database Design)

3.1. Optimize query to find the order with the highest total value

Execute the following query to find the order with the highest total value. Use EXPLAIN ANALYZE to check performance, then propose improvements (e.g., indexes, restructuring).

```
SELECT o.OrderID, SUM(od.Quantity * p.UnitPrice) AS Total
FROM Orders o
JOIN OrdersDetail od ON o.OrderID = od.OrderID
JOIN Product p ON od.ProductID = p.ProductID
GROUP BY o.OrderID
ORDER BY Total DESC
LIMIT 1;
```

3.2. Improve performance of filtering expensive products in specific categories

Compare performance of the query below with and without indexes. Consider changing IN to JOIN and adding indexes on CategoryName and UnitPrice.

```
SELECT ProductName, UnitPrice
FROM Product
WHERE CategoryID IN ( SELECT CategoryID
                      FROM Category
                      WHERE CategoryName ILIKE '%luxury%' )
    AND UnitPrice > 400;
```

3.3. Avoid function in WHERE clause to use indexes:

Avoid function in WHERE clause to use indexes

```
SELECT * FROM Invoice
WHERE DATE_PART('year', InvoiceDate) = 2024;
```

Then rewrite it for index optimization:

```
WHERE InvoiceDate BETWEEN '2024-01-01' AND '2024-12-31';
```

3.4. Evaluate the effect of composite indexes

For the query below, create a composite index and measure performance before and after:

```
SELECT * FROM Orders
WHERE CustomerID = 123 AND OrderDate BETWEEN '2024-01-01' AND '2024-06-30';
```

```
– Index suggestion
CREATE INDEX idx_orders_customer_date ON Orders(CustomerID, OrderDate);
```

Also try reversing the index column order and compare.

3.5. Optimize category-wise inventory aggregation

The following query aggregates inventory by category. Add indexes and evaluate performance using EXPLAIN.

```
SELECT c.CategoryName, SUM(i.QuantityInStock) AS TotalStock
FROM Category c
    JOIN Product p ON c.CategoryID = p.CategoryID
    JOIN Inventory i ON p.ProductID = i.ProductID
GROUP BY c.CategoryName;
```

Try indexes on Product(CategoryID) and Inventory(ProductID).

3.6. Use a Materialized View for Precomputed Aggregation

Calculate total revenue per customer, store it in a materialized view, and compare performance:

```
– Base query
SELECT c.CustomerID, c.CustomerName, SUM(i.TotalAmount) AS Revenue
FROM Customer c
    JOIN Orders o ON c.CustomerID = o.CustomerID
    JOIN Invoice i ON o.OrderID = i.OrderID
GROUP BY c.CustomerID, c.CustomerName;
```

```
– Materialized view
CREATE MATERIALIZED VIEW CustomerRevenue AS
SELECT c.CustomerID, c.CustomerName, SUM(i.TotalAmount) AS Revenue
FROM Customer c
    JOIN Orders o ON c.CustomerID = o.CustomerID
    JOIN Invoice i ON o.OrderID = i.OrderID
GROUP BY c.CustomerID, c.CustomerName;
```

```
– Query from view
SELECT * FROM CustomerRevenue WHERE Revenue > 10000;
```

When should you use materialized views?

3.7. Compare CTE vs Subquery Performance

Evaluate the difference in performance between the following two queries:

```
– CTE
WITH OrderTotal AS ( SELECT o.OrderID, SUM(od.Quantity * p.UnitPrice) AS Total
FROM Orders o
    JOIN OrdersDetail od ON o.OrderID = od.OrderID
    JOIN Product p ON od.ProductID = p.ProductID
GROUP BY o.OrderID
)
SELECT * FROM OrderTotal WHERE Total > 1000;
```

```
– Subquery
SELECT * FROM (
SELECT o.OrderID, SUM(od.Quantity * p.UnitPrice) AS Total
FROM Orders o
    JOIN OrdersDetail od ON o.OrderID = od.OrderID
    JOIN Product p ON od.ProductID = p.ProductID
```

```
GROUP BY o.OrderID
) AS sub
WHERE Total > 1000;
```

Use EXPLAIN ANALYZE and explain when each form is preferred.

3.8. Optimize a Query with Nested JOINs and Filters

Refactor and optimize the following query. It retrieves orders for ”Electronics” products with total quantity > 10:

```
SELECT o.OrderID, c.CustomerName, SUM(od.Quantity) AS TotalQuantity
FROM Orders o
JOIN Customer c ON o.CustomerID = c.CustomerID
JOIN OrdersDetail od ON o.OrderID = od.OrderID
JOIN (
SELECT ProductID FROM Product
WHERE CategoryID IN (
SELECT CategoryID FROM Category WHERE CategoryName = 'Electronics'
)
) AS filtered_products ON od.ProductID = filtered_products.ProductID
GROUP BY o.OrderID, c.CustomerName
HAVING SUM(od.Quantity) > 10;
```

- Can the query be simplified?
- Does indexing Category(CategoryName) or Product(CategoryID) help?
- What's the actual execution plan?

————————-With questions below:

- Draw the algebra tree before and after optimization.
- Compare the theoretical cost based on the number of intermediate tuples.
- Execute both the original and the optimized versions in PostgreSQL using EXPLAIN ANALYZE to measure performance.

3.9. Apply Selection Pushdown to Improve Performance

Original Query:

```
SELECT * FROM Orders o
    JOIN OrdersDetail od ON o.OrderID = od.OrderID
WHERE o.OrderDate >= '2024-01-01' AND od.Quantity > 5;
```

Express the query in **relational algebra** and optimize it using **selection pushdown**. Draw both the **unoptimized** and **optimized algebra trees**.

3.10. Optimize Join Order Based on Selectivity

```
SELECT c.CustomerName, o.OrderID
FROM Customer c
JOIN Orders o ON c.CustomerID = o.CustomerID
JOIN OrdersDetail od ON o.OrderID = od.OrderID
WHERE od.Quantity > 20;
```

Rewrite the query plan by reordering joins to reduce intermediate results. Justify your new join order using selectivity estimation.

3.11. Eliminate Redundant Join

```
SELECT o.OrderID, c.CustomerName
FROM Orders o
    JOIN OrdersDetail od ON o.OrderID = od.OrderID
    JOIN Customer c ON o.CustomerID = c.CustomerID;
```

- Determine if OrdersDetail is necessary to compute the result.
- If it is not used in the SELECT or WHERE, eliminate it and show the optimized relational algebra.

3.12. Projection Pushdown to Minimize Tuple Width

```
SELECT p.ProductName, od.Quantity
FROM OrdersDetail od
JOIN Product p ON od.ProductID = p.ProductID
WHERE od.Quantity > 50;
```

Rewrite the **relational algebra** to **push down projections early** to minimize data volume passed between operations

3.13. Rewrite Nested Subquery as Join and Optimize

```
SELECT p.ProductName
FROM Product p
WHERE p.ProductID IN ( SELECT ProductID
                       FROM OrdersDetail
                       WHERE Quantity > 50 );
```

Rewrite this query as a JOIN, then optimize using **selection pushdown** and **projection pushdown**.

5 Relational Database Schema: Vietnam Geographic.

1. **Country (CountryID, CountryName)**
   Predicate: Each country **(Country)** is assigned a unique code **(CountryID)** to distinguish it from other countries. The country name **(CountryName)** is known.

2. **Province (ProvinceID, ProvinceName, Population, Area, CountryID)**
   Predicate: Each province **(Province)** is assigned a unique code **(ProvinceID)** to distinguish it from other provinces. The province's name **(ProvinceName)**, population **(Population)**, area **(Area)**, and country **(CountryID)** are known.

3. **Border (ProvinceID, NationID)**
   Predicate: The Border relational schema **(Border)** stores the border relationships between provinces **(ProvinceID)** and nations **(NationID)**.

4. **Neighbor (ProvinceID, NeighborID)**
   Predicate: The Neighbor relational schema **(Neighbor)** stores the neighbor relationships between provinces **(ProvinceID)** and neighboring provinces **(NeighborID)**.

Require

1. Find all keys of the Relation Schemas.
2. Create database **DB05**.
3. Create the Relation Schemas.
4. Insert all the required data for queries and integrity constraints.
5. **Performing Queries with Relational Algebra and SQL:**
   - 5.1. Provinces having an area greater than 15,000 square kilometers.
   - 5.2. Provinces (*) that neighbor provinces with an area larger than 15,000 square kilometers.
   - 5.3. Provinces (*) within the country 'North'?
   - 5.4. Which nation borders the northern provinces?
   - 5.5. Calculate the average area of the southern provinces.
   - 5.6. Calculate the population density of the central country.
   - 5.7. Identify the provinces with the highest population density.
   - 5.8. Which provinces have the greatest area?
   - 5.9. In the southern country, provinces with the largest area.
   - 5.10. Provinces that share borders with two or more nations.
   - 5.11. List of countries, showing the number of provinces each has.
   - 5.12. Provinces with the greatest number of neighboring provinces.
   - 5.13. Provinces whose area is larger than the area of their neighboring provinces.
   - 5.14. For each country, list the provinces with the largest areas.
   - 5.15. For each country, list the provinces with a population larger than the country's average population.
   - 5.16. Countries with the greatest total area.
   - 5.17. Countries with the largest total population.
6. **Show all the integrity constraints:**
7. **Physical - Database Design**
   Create new Database abow: **DB06** with thw same struct but with new provinces 34. Using data from **DB05**.