14

Databases

If you profess knowledge of databases, you might be asked some questions on it. We'll review some of the key concepts and offer an overview of how to approach these problems. As you read these queries, don't be surprised by minor variations in syntax. There are a variety of flavors of SQL, and you might have worked with a slightly different one. The examples in this book have been tested against Microsoft SQL Server.

SQL Syntax and Variations

Implicit and explicit joins are shown below. These two statements are equivalent, and it's a matter of personal preference which one you choose. For consistency, we will stick to the explicit join.

Explicit Join

```
1  SELECT CourseName, TeacherName
2  FROM Courses INNER JOIN Teachers
3  ON Courses.TeacherID = Teachers.TeacherID
```

Implicit Join

```
1  SELECT CourseName, TeacherName
2  FROM   Courses, Teachers
3  WHERE  Courses.TeacherID =
4         Teachers.TeacherID
```

Denormalized vs. Normalized Databases

Normalized databases are designed to minimize redundancy, while denormalized databases are designed to optimize read time.

In a traditional normalized database with data like `Courses` and `Teachers`, `Courses` might contain a column called `TeacherID`, which is a foreign key to `Teacher`. One benefit of this is that information about the teacher (name, address, etc.) is only stored once in the database. The drawback is that many common queries will require expensive joins.

Instead, we can denormalize the database by storing redundant data. For example, if we knew that we would have to repeat this query often, we might store the teacher's name in the `Courses` table. Denormalization is commonly used to create highly scalable systems.

SQL Statements

Let's walk through a review of basic SQL syntax, using as an example the database that was mentioned earlier. This database has the following simple structure (* indicates a primary key):

```
Courses: CourseID*, CourseName, TeacherID
Teachers: TeacherID*, TeacherName
Students: StudentID*, StudentName
```

```
StudentCourses: CourseID*, StudentID*
```

Using the above table, implement the following queries.

Query 1: Student Enrollment

Implement a query to get a list of all students and how many courses each student is enrolled in.

At first, we might try something like this:

```
1  /* Incorrect Code */
2  SELECT Students.StudentName, count(*)
3  FROM Students INNER JOIN StudentCourses
4  ON Students.StudentID = StudentCourses.StudentID
5  GROUP BY Students.StudentID
```

This has three problems:

1. We have excluded students who are not enrolled in any courses, since `StudentCourses` only includes enrolled students. We need to change this to a `LEFT JOIN`.
2. Even if we changed it to a `LEFT JOIN`, the query is still not quite right. Doing `count(*)` would return how many items there are in a given group of `StudentID`s. Students enrolled in zero courses would still have one item in their group. We need to change this to count the number of `CourseID`s in each group: `count(StudentCourses.CourseID)`.
3. We've grouped by `Students.StudentID`, but there are still multiple `StudentName`s in each group. How will the database know which `StudentName` to return? Sure, they may all have the same value, but the database doesn't understand that. We need to apply an *aggregate* function to this, such as `first(Students.StudentName)`.

Fixing these issues gets us to this query:

```
1  /* Solution 1: Wrap with another query */
2  SELECT StudentName, Students.StudentID, Cnt
3  FROM (
4    SELECT  Students.StudentID, count(StudentCourses.CourseID) as [Cnt]
5    FROM Students LEFT JOIN StudentCourses
6    ON Students.StudentID = StudentCourses.StudentID
7    GROUP BY Students.StudentID
8  ) T INNER JOIN Students on T.studentID = Students.StudentID
```

Looking at this code, one might ask why we don't just select the student name on line 3 to avoid having to wrap lines 3 through 6 with another query. This (incorrect) solution is shown below.

```
1  /* Incorrect Code */
1  SELECT StudentName, Students.StudentID, count(StudentCourses.CourseID) as [Cnt]
2  FROM Students LEFT JOIN StudentCourses
3  ON Students.StudentID = StudentCourses.StudentID
4  GROUP BY Students.StudentID
```

The answer is that we *can't* do that - at least not exactly as shown. We can only select values that are in an aggregate function or in the GROUP BY clause.

Alternatively, we could resolve the above issues with either of the following statements:

```
1  /* Solution 2: Add StudentName to GROUP BY clause. */
2  SELECT StudentName, Students.StudentID, count(StudentCourses.CourseID) as [Cnt]
3  FROM Students LEFT JOIN StudentCourses
4  ON Students.StudentID = StudentCourses.StudentID
5  GROUP BY Students.StudentID, Students.StudentName
```

OR

```
/* Solution 3: Wrap with aggregate function. */
SELECT  max(StudentName) as [StudentName], Students.StudentID,
        count(StudentCourses.CourseID) as [Count]
FROM Students LEFT JOIN StudentCourses
ON Students.StudentID = StudentCourses.StudentID
GROUP BY Students.StudentID
```

Query 2: Teacher Class Size

Implement a query to get a list of all teachers and how many students they each teach. If a teacher teaches the same student in two courses, you should double count the student. Sort the list in descending order of the number of students a teacher teaches.

We can construct this query step by step. First, let's get a list of `TeacherID`s and how many students are associated with each `TeacherID`. This is very similar to the earlier query.

```
SELECT TeacherID, count(StudentCourses.CourseID) AS [Number]
FROM Courses INNER JOIN StudentCourses
ON Courses.CourseID = StudentCourses.CourseID
GROUP BY Courses.TeacherID
```

Note that this `INNER JOIN` will not select teachers who aren't teaching classes. We'll handle that in the below query when we join it with the list of all teachers.

```
SELECT TeacherName, isnull(StudentSize.Number, 0)
FROM Teachers LEFT JOIN
     (SELECT TeacherID, count(StudentCourses.CourseID) AS [Number]
      FROM Courses INNER JOIN StudentCourses
      ON Courses.CourseID = StudentCourses.CourseID
      GROUP BY Courses.TeacherID) StudentSize
ON Teachers.TeacherID = StudentSize.TeacherID
ORDER BY StudentSize.Number DESC
```

Note how we handled the `NULL` values in the `SELECT` statement to convert the `NULL` values to zeros.

Small Database Design

Additionally, you might be asked to design your own database. We'll walk you through an approach for this. You might notice the similarities between this approach and the approach for object-oriented design.

Step 1: Handle Ambiguity

Database questions often have some ambiguity, intentionally or unintentionally. Before you proceed with your design, you must understand exactly what you need to design.

Imagine you are asked to design a system to represent an apartment rental agency. You will need to know whether this agency has multiple locations or just one. You should also discuss with your interviewer how general you should be. For example, it would be extremely rare for a person to rent two apartments in the same building. But does that mean you shouldn't be able to handle that? Maybe, maybe not. Some very rare conditions might be best handled through a work around (like duplicating the person's contact information in the database).

Step 2: Define the Core Objects

Next, we should look at the core objects of our system. Each of these core objects typically translates into a table. In this case, our core objects might be `Property`, `Building`, `Apartment`, `Tenant` and `Manager`.

Step 3: Analyze Relationships

Outlining the core objects should give us a good sense of what the tables should be. How do these tables relate to each other? Are they many-to-many? One-to-many?

If `Buildings` has a one-to-many relationship with `Apartments` (one `Building` has many `Apartments`), then we might represent this as follows:

Apartments	
ApartmentID	int
ApartmentAddress	varchar(100)
BuildingID	int

Buildings	
BuildingID	int
BuildingName	varchar(100)
BuildingAddress	varchar(500)

Note that the `Apartments` table links back to `Buildings` with a `BuildingID` column.

If we want to allow for the possibility that one person rents more than one apartment, we might want to implement a many-to-many relationship as follows:

TenantApartments	
TenantID	int
ApartmentID	int

Apartments	
ApartmentID	int
ApartmentAddress	varchar(500)
BuildingID	int

Tenants	
TenantID	int
TenantName	varchar(100)
TenantAddress	varchar(500)

The `TenantApartments` table stores a relationship between `Tenants` and `Apartments`.

Step 4: Investigate Actions

Finally, we fill in the details. Walk through the common actions that will be taken and understand how to store and retrieve the relevant data. We'll need to handle lease terms, moving out, rent payments, etc. Each of these actions requires new tables and columns.

▸ Large Database Design

When designing a large, scalable database, joins (which are required in the above examples) are generally very slow. Thus, you must *denormalize* your data. Think carefully about how data will be used—you'll probably need to duplicate the data in multiple tables.

Interview Questions

Questions 1 through 3 refer to the database schema at the end of the chapter. Each apartment can have multiple tenants, and each tenant can have multiple apartments. Each apartment belongs to one building, and each building belongs to one complex.

14.1 **Multiple Apartments:** Write a SQL query to get a list of tenants who are renting more than one apartment.

Hints: #408

pg 441

14.2 **Open Requests:** Write a SQL query to get a list of all buildings and the number of open requests (`Requests` in which `status` equals `'Open'`).

Hints: #411

pg 442

14.3 **Close All Requests:** Building #11 is undergoing a major renovation. Implement a query to close all requests from apartments in this building.

Hints: #431

pg 442

14.4 **Joins:** What are the different types of joins? Please explain how they differ and why certain types are better in certain situations.

Hints: #451

pg 442

14.5 **Denormalization:** What is denormalization? Explain the pros and cons.

Hints: #444, #455

pg 443

14.6 **Entity-Relationship Diagram:** Draw an entity-relationship diagram for a database with companies, people, and professionals (people who work for companies).

Hints: #436

pg 444

14.7 **Design Grade Database:** Imagine a simple database storing information for students' grades. Design what this database might look like and provide a SQL query to return a list of the honor roll students (top 10%), sorted by their grade point average.

Hints: #428, #442

pg 445

Additional Questions: Object-Oriented Design (#7.7), System Design and Scalability (#9.6)

Hints start on page 676.

Apartments	
AptID	int
UnitNumber	varchar(10)
BuildingID	int

Buildings	
BuildingID	int
ComplexID	int
BuildingName	varchar(100)
Address	varchar(500)

Requests	
RequestID	int
Status	varchar(100)
AptID	int
Description	varchar(500)

Complexes	
ComplexID	int
ComplexName	varchar(100)

AptTenants	
TenantID	int
AptID	int

Tenants	
TenantID	int
TenantName	varchar(100)