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)

15

Threads and Locks

In a Microsoft, Google or Amazon interview, it's not terribly common to be asked to implement an algorithm with threads (unless you're working in a team for which this is a particularly important skill). It is, however, relatively common for interviewers at any company to assess your general understanding of threads, particularly your understanding of deadlocks.

This chapter will provide an introduction to this topic.

Threads in Java

Every thread in Java is created and controlled by a unique object of the `java.lang.Thread` class. When a standalone application is run, a user thread is automatically created to execute the `main()` method. This thread is called the main thread.

In Java, we can implement threads in one of two ways:

- By implementing the `java.lang.Runnable` interface
- By extending the `java.lang.Thread` class

We will cover both of these below.

Implementing the Runnable Interface

The `Runnable` interface has the following very simple structure.

```
1  public interface Runnable {
2    void run();
3  }
```

To create and use a thread using this interface, we do the following:

1. Create a class which implements the `Runnable` interface. An object of this class is a `Runnable` object.
2. Create an object of type `Thread` by passing a `Runnable` object as argument to the `Thread` constructor. The `Thread` object now has a `Runnable` object that implements the `run()` method.
3. The `start()` method is invoked on the `Thread` object created in the previous step.

For example:

```
1  public class RunnableThreadExample implements Runnable {
2    public int count = 0;
3
4    public void run() {
5      System.out.println("RunnableThread starting.");
```

```
6         try {
7            while (count < 5) {
8               Thread.sleep(500);
9               count++;
10           }
11        } catch (InterruptedException exc) {
12           System.out.println("RunnableThread interrupted.");
13        }
14        System.out.println("RunnableThread terminating.");
15     }
16  }
17
18  public static void main(String[] args) {
19     RunnableThreadExample instance = new RunnableThreadExample();
20     Thread thread = new Thread(instance);
21     thread.start();
22
23     /* waits until above thread counts to 5 (slowly) */
24     while (instance.count != 5) {
25        try {
26           Thread.sleep(250);
27        } catch (InterruptedException exc) {
28           exc.printStackTrace();
29        }
30     }
31  }
```

In the above code, observe that all we really needed to do is have our class implement the `run()` method (line 4). Another method can then pass an instance of the class to `new Thread(obj)` (lines 19 - 20) and call `start()` on the thread (line 21).

Extending the Thread Class

Alternatively, we can create a thread by extending the `Thread` class. This will almost always mean that we override the `run()` method, and the subclass may also call the thread constructor explicitly in its constructor.

The below code provides an example of this.

```
1  public class ThreadExample extends Thread {
2     int count = 0;
3
4     public void run() {
5        System.out.println("Thread starting.");
6        try {
7           while (count < 5) {
8              Thread.sleep(500);
9              System.out.println("In Thread, count is " + count);
10             count++;
11          }
12       } catch (InterruptedException exc) {
13          System.out.println("Thread interrupted.");
14       }
15       System.out.println("Thread terminating.");
16    }
17 }
18
```

```
19 public class ExampleB {
20   public static void main(String args[]) {
21     ThreadExample instance = new ThreadExample();
22     instance.start();
23
24     while (instance.count != 5) {
25       try {
26         Thread.sleep(250);
27       } catch (InterruptedException exc) {
28         exc.printStackTrace();
29       }
30     }
31   }
32 }
```

This code is very similar to the first approach. The difference is that since we are extending the `Thread` class, rather than just implementing an interface, we can call `start()` on the instance of the class itself.

Extending the Thread Class vs. Implementing the Runnable Interface

When creating threads, there are two reasons why implementing the `Runnable` interface may be preferable to extending the `Thread` class:

- Java does not support multiple inheritance. Therefore, extending the `Thread` class means that the subclass cannot extend any other class. A class implementing the `Runnable` interface will be able to extend another class.
- A class might only be interested in being runnable, and therefore, inheriting the full overhead of the `Thread` class would be excessive.

Synchronization and Locks

Threads within a given process share the same memory space, which is both a positive and a negative. It enables threads to share data, which can be valuable. However, it also creates the opportunity for issues when two threads modify a resource at the same time. Java provides synchronization in order to control access to shared resources.

The keyword `synchronized` and the `lock` form the basis for implementing synchronized execution of code.

Synchronized Methods

Most commonly, we restrict access to shared resources through the use of the `synchronized` keyword. It can be applied to methods and code blocks, and restricts multiple threads from executing the code simultaneously *on the same object*.

To clarify the last point, consider the following code:

```
1  public class MyClass extends Thread  {
2    private String name;
3    private MyObject myObj;
4
5    public MyClass(MyObject obj, String n) {
6      name = n;
7      myObj = obj;
8    }
```

```
9
10    public void run() {
11       myObj.foo(name);
12    }
13 }
14
15 public class MyObject {
16    public synchronized void foo(String name) {
17       try {
18          System.out.println("Thread " + name + ".foo(): starting");
19          Thread.sleep(3000);
20          System.out.println("Thread " + name + ".foo(): ending");
21       } catch (InterruptedException exc) {
22          System.out.println("Thread " + name + ": interrupted.");
23       }
24    }
25 }
```

Can two instances of `MyClass` call `foo` at the same time? It depends. If they have the same instance of `MyObject`, then no. But, if they hold different references, then the answer is yes.

```
1  /* Difference references - both threads can call MyObject.foo() */
2  MyObject obj1 = new MyObject();
3  MyObject obj2 = new MyObject();
4  MyClass thread1 = new MyClass(obj1, "1");
5  MyClass thread2 = new MyClass(obj2, "2");
6  thread1.start();
7  thread2.start()
8
9  /* Same reference to obj. Only one will be allowed to call foo,
10  * and the other will be forced to wait. */
11 MyObject obj = new MyObject();
12 MyClass thread1 = new MyClass(obj, "1");
13 MyClass thread2 = new MyClass(obj, "2");
14 thread1.start()
15 thread2.start()
```

Static methods synchronize on the *class lock*. The two threads above could not simultaneously execute synchronized static methods on the same class, even if one is calling `foo` and the other is calling `bar`.

```
1  public class MyClass extends Thread  {
2     ...
3     public void run() {
4        if (name.equals("1")) MyObject.foo(name);
5        else if (name.equals("2")) MyObject.bar(name);
6     }
7  }
8
9  public class MyObject {
10    public static synchronized void foo(String name) { /* same as before */ }
11    public static synchronized void bar(String name) { /* same as foo */ }
12 }
```

If you run this code, you will see the following printed:

```
Thread 1.foo(): starting
Thread 1.foo(): ending
Thread 2.bar(): starting
Thread 2.bar(): ending
```

Synchronized Blocks

Similarly, a block of code can be synchronized. This operates very similarly to synchronizing a method.

```
1  public class MyClass extends Thread  {
2    ...
3    public void run() {
4      myObj.foo(name);
5    }
6  }
7  public class MyObject {
8    public void foo(String name) {
9      synchronized(this) {
10       ...
11     }
12   }
13 }
```

Like synchronizing a method, only one thread per instance of `MyObject` can execute the code within the `synchronized` block. That means that, if `thread1` and `thread2` have the same instance of `MyObject`, only one will be allowed to execute the code block at a time.

Locks

For more granular control, we can utilize a lock. A lock (or monitor) is used to synchronize access to a shared resource by associating the resource with the lock. A thread gets access to a shared resource by first acquiring the lock associated with the resource. At any given time, at most one thread can hold the lock and, therefore, only one thread can access the shared resource.

A common use case for locks is when a resource is accessed from multiple places, but should be only accessed by one thread *at a time*. This case is demonstrated in the code below.

```
1  public class LockedATM {
2    private Lock lock;
3    private int balance = 100;
4
5    public LockedATM() {
6      lock = new ReentrantLock();
7    }
8
9    public int withdraw(int value) {
10     lock.lock();
11     int temp = balance;
12     try {
13       Thread.sleep(100);
14       temp = temp - value;
15       Thread.sleep(100);
16       balance = temp;
17     } catch (InterruptedException e) {     }
18     lock.unlock();
19     return temp;
20   }
21
22   public int deposit(int value) {
23     lock.lock();
24     int temp = balance;
25     try {
26       Thread.sleep(100);
```

```
27          temp = temp + value;
28          Thread.sleep(300);
29          balance = temp;
30        } catch (InterruptedException e) {     }
31        lock.unlock();
32        return temp;
33      }
34  }
```

Of course, we've added code to intentionally slow down the execution of `withdraw` and `deposit`, as it helps to illustrate the potential problems that can occur. You may not write code exactly like this, but the situation it mirrors is very, very real. Using a lock will help protect a shared resource from being modified in unexpected ways.

▸ Deadlocks and Deadlock Prevention

A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds (or an equivalent situation with several threads). Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever. The threads are said to be deadlocked.

In order for a deadlock to occur, you must have all four of the following conditions met:

1. *Mutual Exclusion:* Only one process can access a resource at a given time. (Or, more accurately, there is limited access to a resource. A deadlock could also occur if a resource has limited quantity.)
2. *Hold and Wait:* Processes already holding a resource can request additional resources, without relinquishing their current resources.
3. *No Preemption:* One process cannot forcibly remove another process' resource.
4. *Circular Wait:* Two or more processes form a circular chain where each process is waiting on another resource in the chain.

Deadlock prevention entails removing any of the above conditions, but it gets tricky because many of these conditions are difficult to satisfy. For instance, removing #1 is difficult because many resources can only be used by one process at a time (e.g., printers). Most deadlock prevention algorithms focus on avoiding condition #4: circular wait.

Interview Questions

15.1 **Thread vs. Process:** What's the difference between a thread and a process?

Hints: #405

pg 447

15.2 **Context Switch:** How would you measure the time spent in a context switch?

Hints: #403, #407, #415, #441

pg 447

15.3 **Dining Philosophers:** In the famous dining philosophers problem, a bunch of philosophers are sitting around a circular table with one chopstick between each of them. A philosopher needs both chopsticks to eat, and always picks up the left chopstick before the right one. A deadlock could potentially occur if all the philosophers reached for the left chopstick at the same time. Using threads and locks, implement a simulation of the dining philosophers problem that prevents deadlocks.

Hints: #419, #437

pg 449

15.4 **Deadlock-Free Class:** Design a class which provides a lock only if there are no possible deadlocks.

Hints: #422, #434

pg 452

15.5 **Call In Order:** Suppose we have the following code:

```
public class Foo {
   public Foo() { ... }
   public void first() { ... }
   public void second() { ... }
   public void third() { ... }
}
```

The same instance of `Foo` will be passed to three different threads. `ThreadA` will call `first`, `threadB` will call `second`, and `threadC` will call `third`. Design a mechanism to ensure that `first` is called before `second` and `second` is called before `third`.

Hints: #417, #433, #446

pg 456

15.6 **Synchronized Methods:** You are given a class with synchronized method A and a normal method B. If you have two threads in one instance of a program, can they both execute A at the same time? Can they execute A and B at the same time?

Hints: #429

pg 458

15.7 **FizzBuzz**: In the classic problem FizzBuzz, you are told to print the numbers from 1 to n. However, when the number is divisible by 3, print "Fizz". When it is divisible by 5, print "Buzz". When it is divisible by 3 and 5, print "FizzBuzz". In this problem, you are asked to do this in a multithreaded way. Implement a multithreaded version of FizzBuzz with four threads. One thread checks for divisibility of 3 and prints "Fizz". Another thread is responsible for divisibility of 5 and prints "Buzz". A third thread is responsible for divisibility of 3 and 5 and prints "FizzBuzz". A fourth thread does the numbers.

Hints: #414, #439, #447, #458

pg 458

Hints start on page 676.