Instead, rather than generating a subset based on sizes, let's think about it based on elements. (The fact that we're told to use lambda expressions is also a hint that we should think about some sort of iteration or processing through the elements.)

Imagine we were iterating through `{1, 2, 3}` to generate a subset. Should 1 be in this subset?

We've got two choices: yes or no. We need to weight the probability of "yes" vs. "no" based on the percent of subsets that contain 1. So, what percent of elements contain 1?

For any specific element, there are as many subsets that contain the element as do not contain it. Consider the following:

```
{}          {1}
{2}         {1, 2}
{3}         {1, 3}
{2, 3}      {1, 2, 3}
```

Note how the difference between the subsets on the left and the subsets on the right is the existence of 1. The left and right sides must have the same number of subsets because we can convert from one to the other by just adding an element.

This means that we can generate a random subset by iterating through the list and flipping a coin (i.e., deciding on a 50/50 chance) to pick whether or not each element will be in it.

Without lambda expressions, we can write something like this:

```
1  List<Integer> getRandomSubset(List<Integer> list) {
2    List<Integer> subset = new ArrayList<Integer>();
3    Random random = new Random();
4    for (int item : list) {
5      /* Flip coin. */
6      if (random.nextBoolean()) {
7        subset.add(item);
8      }
9    }
10   return subset;
11 }
```

To implement this approach using lambda expressions, we can do the following:

```
1  List<Integer> getRandomSubset(List<Integer> list) {
2    Random random = new Random();
3    List<Integer> subset = list.stream().filter(
4      k -> { return random.nextBoolean(); /* Flip coin. */
5    }).collect(Collectors.toList());
6    return subset;
7  }
```

Or, we can use a predicate (defined within the class or within the function):

```
1  Random random = new Random();
2  Predicate<Object> flipCoin = o -> {
3    return random.nextBoolean();
4  };
5
6  List<Integer> getRandomSubset(List<Integer> list) {
7    List<Integer> subset = list.stream().filter(flipCoin).
8      collect(Collectors.toList());
9    return subset;
10 }
```

The nice thing about this implementation is that now we can apply the `flipCoin` predicate in other places.

14

Solutions to Databases

Questions 1 through 3 refer to the following database schema:

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)

Note that 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.

pg 172

SOLUTION

To implement this, we can use the `HAVING` and `GROUP BY` clauses and then perform an `INNER JOIN` with `Tenants`.

```
1  SELECT TenantName
2  FROM Tenants
3  INNER JOIN
4     (SELECT TenantID FROM AptTenants GROUP BY TenantID HAVING count(*) > 1) C
5  ON Tenants.TenantID = C.TenantID
```

Whenever you write a `GROUP BY` clause in an interview (or in real life), make sure that anything in the `SELECT` clause is either an aggregate function or contained within the `GROUP BY` clause.

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').

pg 173

SOLUTION

This problem uses a straightforward join of Requests and Apartments to get a list of building IDs and the number of open requests. Once we have this list, we join it again with the Buildings table.

```
1  SELECT BuildingName, ISNULL(Count, 0) as 'Count'
2  FROM Buildings
3  LEFT JOIN
4    (SELECT Apartments.BuildingID, count(*) as 'Count'
5     FROM Requests INNER JOIN Apartments
6     ON Requests.AptID = Apartments.AptID
7     WHERE Requests.Status = 'Open'
8     GROUP BY Apartments.BuildingID) ReqCounts
9  ON ReqCounts.BuildingID = Buildings.BuildingID
```

Queries like this that utilize sub-queries should be thoroughly tested, even when coding by hand. It may be useful to test the inner part of the query first, and then test the outer part.

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

pg 173

SOLUTION

UPDATE queries, like SELECT queries, can have WHERE clauses. To implement this query, we get a list of all apartment IDs within building #11 and the list of update requests from those apartments.

```
1  UPDATE Requests
2  SET Status = 'Closed'
3  WHERE AptID IN (SELECT AptID FROM Apartments WHERE BuildingID = 11)
```

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

pg 173

SOLUTION

JOIN is used to combine the results of two tables. To perform a JOIN, each of the tables must have at least one field that will be used to find matching records from the other table. The join type defines which records will go into the result set.

Let's take for example two tables: one table lists the "regular" beverages, and another lists the calorie-free beverages. Each table has two fields: the beverage name and its product code. The "code" field will be used to perform the record matching.

Regular Beverages:

Name	Code
Budweiser	BUDWEISER
Coca-Cola	COCACOLA

Name	Code
Pepsi	PEPSI

Calorie-Free Beverages:

Name	Code
Diet Coca-Cola	COCACOLA
Fresca	FRESCA
Diet Pepsi	PEPSI
Pepsi Light	PEPSI
Purified Water	Water

If we wanted to join `Beverage` with `Calorie-Free Beverages`, we would have many options. These are discussed below.

- `INNER JOIN`: The result set would contain only the data where the criteria match. In our example, we would get three records: one with a COCACOLA code and two with `PEPSI` codes.
- `OUTER JOIN`: An `OUTER JOIN` will always contain the results of `INNER JOIN`, but it may also contain some records that have no matching record in the other table. `OUTER JOIN`s are divided into the following subtypes:
  - `LEFT OUTER JOIN`, or simply `LEFT JOIN`: The result will contain all records from the left table. If no matching records were found in the right table, then its fields will contain the `NULL` values. In our example, we would get four records. In addition to `INNER JOIN` results, `BUDWEISER` would be listed, because it was in the left table.
  - `RIGHT OUTER JOIN`, or simply `RIGHT JOIN`: This type of join is the opposite of `LEFT JOIN`. It will contain every record from the right table; the missing fields from the left table will be `NULL`. Note that if we have two tables, A and B, then we can say that the statement A `LEFT JOIN` B is equivalent to the statement B `RIGHT JOIN` A. In our example above, we will get five records. In addition to `INNER JOIN` results, `FRESCA` and `WATER` records will be listed.
  - `FULL OUTER JOIN`: This type of join combines the results of the `LEFT` and `RIGHT JOINS`. All records from both tables will be included in the result set, regardless of whether or not a matching record exists in the other table. If no matching record was found, then the corresponding result fields will have a `NULL` value. In our example, we will get six records.

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

pg 173

SOLUTION

Denormalization is a database optimization technique in which we add redundant data to one or more tables. This can help us avoid costly joins in a relational database.

By contrast, in a traditional normalized database, we store data in separate logical tables and attempt to minimize redundant data. We may strive to have only one copy of each piece of data in the database.

For example, in a normalized database, we might have a `Courses` table and a `Teachers` table. Each entry in `Courses` would store the `teacherID` for a `Course` but not the `teacherName`. When we need to retrieve a list of all `Courses` with the `Teacher` name, we would do a join between these two tables.

In some ways, this is great; if a teacher changes his or her name, we only have to update the name in one place.

The drawback, however, is that if the tables are large, we may spend an unnecessarily long time doing joins on tables.

Denormalization, then, strikes a different compromise. Under denormalization, we decide that we're okay with some redundancy and some extra effort to update the database in order to get the efficiency advantages of fewer joins.

Cons of Denormalization	Pros of Denormalization
Updates and inserts are more expensive.	Retrieving data is faster since we do fewer joins.
Denormalization can make update and insert code harder to write.	Queries to retrieve can be simpler (and therefore less likely to have bugs), since we need to look at fewer tables.
Data may be inconsistent. Which is the "correct" value for a piece of data?	
Data redundancy necessitates more storage.	

In a system that demands scalability, like that of any major tech companies, we almost always use elements of both normalized and denormalized databases.

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

pg 173

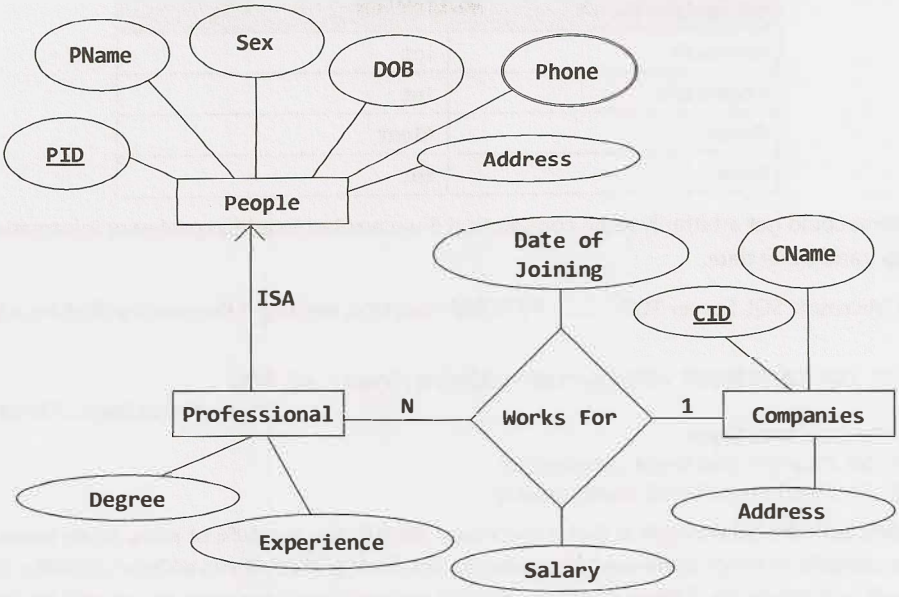
SOLUTION

People who work for `Companies` are `Professionals`. So, there is an `ISA` ("is a") relationship between `People` and `Professionals` (or we could say that a `Professional` is derived from `People`).

Each `Professional` has additional information such as degree and work experiences in addition to the properties derived from `People`.

A `Professional` works for one company at a time (probably—you might want to validate this assumption), but `Companies` can hire many `Professionals`. So, there is a many-to-one relationship between `Professionals` and `Companies`. This "Works For" relationship can store attributes such as an employee's start date and salary. These attributes are defined only when we relate a `Professional` with a `Company`.

A `Person` can have multiple phone numbers, which is why `Phone` is a multi-valued attribute.



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.

pg 173

SOLUTION

In a simplistic database, we'll have at least three objects: `Students`, `Courses`, and `CourseEnrollment`. `Students` will have at least a student name and ID and will likely have other personal information. `Courses` will contain the course name and ID and will likely contain the course description, professor, and other information. `CourseEnrollment` will pair `Students` and `Courses` and will also contain a field for `CourseGrade`.

Students	
StudentID	int
StudentName	varchar(100)
Address	varchar(500)

Courses	
CourseID	int
CourseName	varchar(100)
ProfessorID	int

CourseEnrollment	
CourseID	int
StudentID	int
Grade	float
Term	int

This database could get arbitrarily more complicated if we wanted to add in professor information, billing information, and other data.

Using the Microsoft SQL Server TOP ... PERCENT function, we might (incorrectly) first try a query like this:

```
1  SELECT TOP 10 PERCENT AVG(CourseEnrollment.Grade) AS GPA,
2                                                  CourseEnrollment.StudentID
3  FROM CourseEnrollment
4  GROUP BY CourseEnrollment.StudentID
5  ORDER BY AVG(CourseEnrollment.Grade)
```

The problem with the above code is that it will return literally the top 10% of rows, when sorted by GPA. Imagine a scenario in which there are 100 students, and the top 15 students all have 4.0 GPAs. The above function will only return 10 of those students, which is not really what we want. In case of a tie, we want to include the students who tied for the top 10% -- even if this means that our honor roll includes more than 10% of the class.

To correct this issue, we can build something similar to this query, but instead first get the GPA cut off.

```
1  DECLARE @GPACutOff float;
2  SET @GPACutOff = (SELECT min(GPA) as 'GPAMin' FROM (
3        SELECT TOP 10 PERCENT AVG(CourseEnrollment.Grade) AS GPA
4        FROM CourseEnrollment
5        GROUP BY CourseEnrollment.StudentID
6        ORDER BY GPA desc) Grades);
```

Then, once we have @GPACutOff defined, selecting the students with at least this GPA is reasonably straightforward.

```
1  SELECT StudentName, GPA
2  FROM (SELECT AVG(CourseEnrollment.Grade) AS GPA, CourseEnrollment.StudentID
3        FROM CourseEnrollment
4        GROUP BY CourseEnrollment.StudentID
5        HAVING AVG(CourseEnrollment.Grade) >= @GPACutOff) Honors
6  INNER JOIN Students ON Honors.StudentID = Student.StudentID
```

Be very careful about what implicit assumptions you make. If you look at the above database description, what potentially incorrect assumption do you see? One is that each course can only be taught by one professor. At some schools, courses may be taught by multiple professors.

However, you *will* need to make some assumptions, or you'd drive yourself crazy. Which assumptions you make is less important than just recognizing *that* you made assumptions. Incorrect assumptions, both in the real world and in an interview, can be dealt with *as long as they are acknowledged.*

Remember, additionally, that there's a trade-off between flexibility and complexity. Creating a system in which a course can have multiple professors does increase the database's flexibility, but it also increases its complexity. If we tried to make our database flexible to every possible situation, we'd wind up with something hopelessly complex.

Make your design reasonably flexible, and state any other assumptions or constraints. This goes for not just database design, but object-oriented design and programming in general.

15

Solutions to Threads and Locks

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

pg 179

SOLUTION

Processes and threads are related to each other but are fundamentally different.

A process can be thought of as an instance of a program in execution. A process is an independent entity to which system resources (e.g., CPU time and memory) are allocated. Each process is executed in a separate address space, and one process cannot access the variables and data structures of another process. If a process wishes to access another process' resources, inter-process communications have to be used. These include pipes, files, sockets, and other forms.

A thread exists within a process and shares the process' resources (including its heap space). Multiple threads within the same process will share the same heap space. This is very different from processes, which cannot directly access the memory of another process. Each thread still has its own registers and its own stack, but other threads can read and write the heap memory.

A thread is a particular execution path of a process. When one thread modifies a process resource, the change is immediately visible to sibling threads.

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

pg 179

SOLUTION

This is a tricky question, but let's start with a possible solution.

A context switch is the time spent switching between two processes (i.e., bringing a waiting process into execution and sending an executing process into waiting/terminated state). This happens in multitasking. The operating system must bring the state information of waiting processes into memory and save the state information of the currently running process.

In order to solve this problem, we would like to record the timestamps of the last and first instruction of the swapping processes. The context switch time is the difference in the timestamps between the two processes.

Let's take an easy example: Assume there are only two processes, P_1 and P_2.

P_1 is executing and P_2 is waiting for execution. At some point, the operating system must swap P_1 and P_2—let's assume it happens at the Nth instruction of P_1. If $t_{x,k}$ indicates the timestamp in microseconds of the kth instruction of process x, then the context switch would take $t_{2,1} - t_{1,n}$ microseconds.

The tricky part is this: how do we know when this swapping occurs? We cannot, of course, record the timestamp of every instruction in the process.

Another issue is that swapping is governed by the scheduling algorithm of the operating system and there may be many kernel level threads which are also doing context switches. Other processes could be contending for the CPU or the kernel handling interrupts. The user does not have any control over these extraneous context switches. For instance, if at time $t_{1,n}$ the kernel decides to handle an interrupt, then the context switch time would be overstated.

In order to overcome these obstacles, we must first construct an environment such that after P_1 executes, the task scheduler immediately selects P_2 to run. This may be accomplished by constructing a data channel, such as a pipe, between P_1 and P_2 and having the two processes play a game of ping-pong with a data token.

That is, let's allow P_1 to be the initial sender and P_2 to be the receiver. Initially, P_2 is blocked (sleeping) as it awaits the data token. When P_1 executes, it delivers the token over the data channel to P_2 and immediately attempts to read a response token. However, since P_2 has not yet had a chance to run, no such token is available for P_1 and the process is blocked. This relinquishes the CPU.

A context switch results and the task scheduler must select another process to run. Since P_2 is now in a ready-to-run state, it is a desirable candidate to be selected by the task scheduler for execution. When P_2 runs, the roles of P_1 and P_2 are swapped. P_2 is now acting as the sender and P_1 as the blocked receiver. The game ends when P_2 returns the token to P_1.

To summarize, an iteration of the game is played with the following steps:

1. P_2 blocks awaiting data from P_1.
2. P_1 marks the start time.
3. P_1 sends token to P_2.
4. P_1 attempts to read a response token from P_2. This induces a context switch.
5. P_2 is scheduled and receives the token.
6. P_2 sends a response token to P_1.
7. P_2 attempts read a response token from P_1. This induces a context switch.
8. P_1 is scheduled and receives the token.
9. P_1 marks the end time.

The key is that the delivery of a data token induces a context switch. Let T_d and T_r be the time it takes to deliver and receive a data token, respectively, and let T_c be the amount of time spent in a context switch. At step 2, P_1 records the timestamp of the delivery of the token, and at step 9, it records the timestamp of the response. The amount of time elapsed, T, between these events may be expressed by:

$$T = 2 * (T_d + T_c + T_r)$$

This formula arises because of the following events: P_1 sends a token (3), the CPU context switches (4), P_2 receives it (5). P_2 then sends the response token (6), the CPU context switches (7), and finally P_1 receives it (8).

P_1 will be able to easily compute T, since this is just the time between events 3 and 8. So, to solve for T_c, we must first determine the value of $T_d + T_r$.

How can we do this? We can do this by measuring the length of time it takes P_1 to send and receive a token to itself. This will not induce a context switch since P_1 is running on the CPU at the time it sent the token and will not block to receive it.

The game is played a number of iterations to average out any variability in the elapsed time between steps 2 and 9 that may result from unexpected kernel interrupts and additional kernel threads contending for the CPU. We select the smallest observed context switch time as our final answer.

However, all we can ultimately say that this is an approximation which depends on the underlying system. For example, we make the assumption that P_2 is selected to run once a data token becomes available. However, this is dependent on the implementation of the task scheduler and we cannot make any guarantees.

That's okay; it's important in an interview to recognize when your solution might not be perfect.

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.

pg 180

SOLUTION

First, let's implement a simple simulation of the dining philosophers problem in which we don't concern ourselves with deadlocks. We can implement this solution by having `Philosopher` extend `Thread`, and `Chopstick` call `lock.lock()` when it is picked up and `lock.unlock()` when it is put down.

```
1  class Chopstick {
2    private Lock lock;
3
4    public Chopstick() {
5      lock = new ReentrantLock();
6    }
7
8    public void pickUp() {
9      void lock.lock();
10   }
11
12   public void putDown() {
13     lock.unlock();
14   }
15 }
16
17 class Philosopher extends Thread {
18   private int bites = 10;
19   private Chopstick left, right;
20
21   public Philosopher(Chopstick left, Chopstick right) {
22     this.left = left;
23     this.right = right;
24   }
```