```
    int rightFalse = countEval(right, false);
    int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);

    int totalTrue = 0;
    if (c == '^') { // required: one true and one false
      totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
    } else if (c == '&') { // required: both true
      totalTrue = leftTrue * rightTrue;
    } else if (c == '|') { // required: anything but both false
      totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
                  leftTrue * rightFalse;
    }

    int subWays = result ? totalTrue : total - totalTrue;
    ways += subWays;
  }

  return ways;
}

boolean stringToBool(String c) {
  return c.equals("1") ? true : false;
}
```

Note that the tradeoff of computing the `false` results from the `true` ones, and of computing the `{leftTrue, rightTrue, leftFalse, and rightFalse}` values upfront, is a small amount of extra work in some cases. For example, if we're looking for the ways that an AND (&) can result in `true`, we never would have needed the `leftFalse` and `rightFalse` results. Likewise, if we're looking for the ways that an OR (|) can result in `false`, we never would have needed the `leftTrue` and `rightTrue` results.

Our current code is blind to what we do and don't actually need to do and instead just computes all of the values. This is probably a reasonable tradeoff to make (especially given the constraints of whiteboard coding) as it makes our code substantially shorter and less tedious to write. Whichever approach you make, you should discuss the tradeoffs with your interviewer.

That said, there are more important optimizations we can make.

Optimized Solutions

If we follow the recursive path, we'll note that we end up doing the same computation repeatedly.

Consider the expression `0^0&0^1|1` and these recursion paths:

- Add parens around char 1. `(0)^(0&0^1|1)`
  - Add parens around char 3. `(0)^((0)&(0^1|1))`
- Add parens around char 3. `(0^0)&(0^1|1)`
  - Add parens around char 1. `((0)^(0))&(0^1|1)`

Although these two expressions are different, they have a similar component: `(0^1|1)`. We should reuse our effort on this.

We can do this by using memoization, or a hash table. We just need to store the result of `countEval(expression, result)` for each expression and result. If we see an expression that we've calculated before, we just return it from the cache.

```
int countEval(String s, boolean result, HashMap<String, Integer> memo) {
  if (s.length() == 0) return 0;
  if (s.length() == 1) return stringToBool(s) == result ? 1 : 0;
  if (memo.containsKey(result + s)) return memo.get(result + s);

  int ways = 0;

  for (int i = 1; i < s.length(); i += 2) {
    char c = s.charAt(i);
    String left = s.substring(0, i);
    String right = s.substring(i + 1, s.length());
    int leftTrue = countEval(left, true, memo);
    int leftFalse = countEval(left, false, memo);
    int rightTrue = countEval(right, true, memo);
    int rightFalse = countEval(right, false, memo);
    int total = (leftTrue + leftFalse) * (rightTrue + rightFalse);

    int totalTrue = 0;
    if (c == '^') {
      totalTrue = leftTrue * rightFalse + leftFalse * rightTrue;
    } else if (c == '&') {
      totalTrue = leftTrue * rightTrue;
    } else if (c == '|') {
      totalTrue = leftTrue * rightTrue + leftFalse * rightTrue +
                  leftTrue * rightFalse;
    }

    int subWays = result ? totalTrue : total - totalTrue;
    ways += subWays;
  }

  memo.put(result + s, ways);
  return ways;
}
```

The added benefit of this is that we could actually end up with the same substring in multiple parts of the expression. For example, an expression like `0^1^0&0^1^0` has two instances of `0^1^0`. By caching the result of the substring value in a memoization table, we'll get to reuse the result for the right part of the expression after computing it for the left.

There is one further optimization we can make, but it's far beyond the scope of the interview. There *is* a closed form expression for the number of ways of parenthesizing an expression, but you wouldn't be expected to know it. It is given by the Catalan numbers, where n is the number of operators:

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

We could use this to compute the total ways of evaluating the expression. Then, rather than computing `leftTrue` and `leftFalse`, we just compute one of those and calculate the other using the Catalan numbers. We would do the same thing for the right side.

9

Solutions to System Design and Scalability

9.1 **Stock Data:** Imagine you are building some sort of service that will be called by up to 1,000 client applications to get simple end-of-day stock price information (open, close, high, low). You may assume that you already have the data, and you can store it in any format you wish. How would you design the client-facing service that provides the information to client applications? You are responsible for the development, rollout, and ongoing monitoring and maintenance of the feed. Describe the different methods you considered and why you would recommend your approach. Your service can use any technologies you wish, and can distribute the information to the client applications in any mechanism you choose.

pg 144

SOLUTION

From the statement of the problem, we want to focus on how we actually distribute the information to clients. We can assume that we have some scripts that magically collect the information.

We want to start off by thinking about what the different aspects we should consider in a given proposal are:

- *Client Ease of Use:* We want the service to be easy for the clients to implement and useful for them.
- *Ease for Ourselves:* This service should be as easy as possible for us to implement, as we shouldn't impose unnecessary work on ourselves. We need to consider in this not only the cost of implementing, but also the cost of maintenance.
- *Flexibility for Future Demands:* This problem is stated in a "what would you do in the real world" way, so we should think like we would in a real-world problem. Ideally, we do not want to overly constrain ourselves in the implementation, such that we can't be flexible if the requirements or demands change.
- *Scalability and Efficiency:* We should be mindful of the efficiency of our solution, so as not to overly burden our service.

With this framework in mind, we can consider various proposals.

Proposal #1

One option is that we could keep the data in simple text files and let clients download the data through some sort of FTP server. This would be easy to maintain in some sense, since files can be easily viewed and backed up, but it would require more complex parsing to do any sort of query. And, if additional data were added to our text file, it might break the clients' parsing mechanism.

Proposal #2

We could use a standard SQL database, and let the clients plug directly into that. This would provide the following benefits:

- Facilitates an easy way for the clients to do query processing over the data, in case there are additional features we need to support. For example, we could easily and efficiently perform a query such as "return all stocks having an open price greater than N and a closing price less than M."
- Rolling back, backing up data, and security could be provided using standard database features. We don't have to "reinvent the wheel," so it's easy for us to implement.
- Reasonably easy for the clients to integrate into existing applications. SQL integration is a standard feature in software development environments.

What are the disadvantages of using a SQL database?

- It's much heavier weight than we really need. We don't necessarily need all the complexity of a SQL backend to support a feed of a few bits of information.
- It's difficult for humans to be able to read it, so we'll likely need to implement an additional layer to view and maintain the data. This increases our implementation costs.
- Security: While a SQL database offers pretty well defined security levels, we would still need to be very careful to not give clients access that they shouldn't have. Additionally, even if clients aren't doing anything "malicious," they might perform expensive and inefficient queries, and our servers would bear the costs of that.

These disadvantages don't mean that we shouldn't provide SQL access. Rather, they mean that we should be aware of the disadvantages.

Proposal #3

XML is another great option for distributing the information. Our data has fixed format and fixed size: `company_name`, `open`, `high`, `low`, `closing price`. The XML could look like this:

```
<root>
  <date value="2008-10-12">
    <company name="foo">
      <open>126.23</open>
      <high>130.27</high>
      <low>122.83</low>
      <closingPrice>127.30</closingPrice>
    </company>
    <company name="bar">
      <open>52.73</open>
      <high>60.27</high>
      <low>50.29</low>
      <closingPrice>54.91</closingPrice>
    </company>
  </date>
  <date value="2008-10-11"> . . . </date>
</root>
```

The advantages of this approach include the following:

- It's very easy to distribute, and it can also be easily read by both machines and humans. This is one reason that XML is a standard data model to share and distribute data.
- Most languages have a library to perform XML parsing, so it's reasonably easy for clients to implement.

- We can add new data to the XML file by adding additional nodes. This would not break the client's parser (provided they have implemented their parser in a reasonable way).
- Since the data is being stored as XML files, we can use existing tools for backing up the data. We don't need to implement our own backup tool.

The disadvantages may include:

- This solution sends the clients all the information, even if they only want part of it. It is inefficient in that way.
- Performing any queries on the data requires parsing the entire file.

Regardless of which solution we use for data storage, we could provide a web service (e.g., SOAP) for client data access. This adds a layer to our work, but it can provide additional security, and it may even make it easier for clients to integrate the system.

However—and this is a pro and a con—clients will be limited to grabbing the data only how we expect or want them to. By contrast, in a pure SQL implementation, clients could query for the highest stock price, even if this wasn't a procedure we "expected" them to need.

So which one of these would we use? There's no clear answer. The pure text file solution is probably a bad choice, but you can make a compelling argument for the SQL or XML solution, with or without a web service.

The goal of a question like this is not to see if you get the "correct" answer (there is no single correct answer). Rather, it's to see how you design a system, and how you evaluate trade-offs.

9.2 **Social Network:** How would you design the data structures for a very large social network like Facebook or LinkedIn? Describe how you would design an algorithm to show the shortest path between two people (e.g., Me -> Bob -> Susan -> Jason -> You).

pg 145

SOLUTION

A good way to approach this problem is to remove some of the constraints and solve it for that situation first.

Step 1: Simplify the Problem—Forget About the Millions of Users

First, let's forget that we're dealing with millions of users. Design this for the simple case.

We can construct a graph by treating each person as a node and letting an edge between two nodes indicate that the two users are friends.

If I wanted to find the path between two people, I could start with one person and do a simple breadth-first search.

Why wouldn't a depth-first search work well? First, depth-first search would just find a path. It wouldn't necessarily find the shortest path. Second, even if we just needed any path, it would be very inefficient. Two users might be only one degree of separation apart, but I could search millions of nodes in their "subtrees" before finding this relatively immediate connection.

Alternatively, I could do what's called a bidirectional breadth-first search. This means doing two breadth-first searches, one from the source and one from the destination. When the searches collide, we know we've found a path.

In the implementation, we'll use two classes to help us. `BFSData` holds the data we need for a breadth-first search, such as the `isVisited` hash table and the `toVisit` queue. `PathNode` will represent the path as we're searching it, storing each `Person` and the `previousNode` we visited in this path.

```
LinkedList<Person> findPathBiBFS(HashMap<Integer, Person> people, int source,
                                  int destination) {
  BFSData sourceData = new BFSData(people.get(source));
  BFSData destData = new BFSData(people.get(destination));

  while (!sourceData.isFinished() && !destData.isFinished()) {
    / *Search out from source. */
    Person collision = searchLevel(people, sourceData, destData);
    if (collision != null) {
      return mergePaths(sourceData, destData, collision.getID());
    }

    / *Search out from destination. */
    collision = searchLevel(people, destData, sourceData);
    if (collision != null) {
      return mergePaths(sourceData, destData, collision.getID());
    }
  }
  return null;
}

/ *Search one level and return collision, if any. */
Person searchLevel(HashMap<Integer, Person> people, BFSData primary,
                   BFSData secondary) {
  / *We only want to search one level at a time. Count how many nodes are
   * currently in the primary's level and only do that many nodes. We'll continue
   * to add nodes to the end. */
  int count = primary.toVisit.size();
  for (int i = 0; i < count; i++) {
    / *Pull out first node. */
    PathNode pathNode = primary.toVisit.poll();
    int personId = pathNode.getPerson().getID();

    / *Check if it's already been visited. */
    if (secondary.visited.containsKey(personId)) {
      return pathNode.getPerson();
    }

    / *Add friends to queue. */
    Person person = pathNode.getPerson();
    ArrayList<Integer> friends = person.getFriends();
    for (int friendId : friends) {
      if (!primary.visited.containsKey(friendId)) {
        Person friend = people.get(friendId);
        PathNode next = new PathNode(friend, pathNode);
        primary.visited.put(friendId, next);
        primary.toVisit.add(next);
      }
    }
  }
  return null;
}

```

```
/* Merge paths where searches met at connection. */
LinkedList<Person> mergePaths(BFSData bfs1, BFSData bfs2, int connection) {
  PathNode end1 = bfs1.visited.get(connection); // end1 -> source
  PathNode end2 = bfs2.visited.get(connection); // end2 -> dest
  LinkedList<Person> pathOne = end1.collapse(false);
  LinkedList<Person> pathTwo = end2.collapse(true); // reverse
  pathTwo.removeFirst(); // remove connection
  pathOne.addAll(pathTwo); // add second path
  return pathOne;
}

class PathNode {
  private Person person = null;
  private PathNode previousNode = null;
  public PathNode(Person p, PathNode previous) {
    person = p;
    previousNode = previous;
  }

  public Person getPerson() { return person; }

  public LinkedList<Person> collapse(boolean startsWithRoot) {
    LinkedList<Person> path = new LinkedList<Person>();
    PathNode node = this;
    while (node != null) {
      if (startsWithRoot) {
        path.addLast(node.person);
      } else {
        path.addFirst(node.person);
      }
      node = node.previousNode;
    }
    return path;
  }
}

class BFSData {
  public Queue<PathNode> toVisit = new LinkedList<PathNode>();
  public HashMap<Integer, PathNode> visited =
    new HashMap<Integer, PathNode>();

  public BFSData(Person root) {
    PathNode sourcePath = new PathNode(root, null);
    toVisit.add(sourcePath);
    visited.put(root.getID(), sourcePath);
  }

  public boolean isFinished() {
    return toVisit.isEmpty();
  }
}
```

Many people are surprised that this is faster. Some quick math can explain why.

Suppose every person has k friends, and node S and node D have a friend C in common.

- Traditional breadth-first search from S to D: We go through roughly k+k*k nodes: each of S's k friends, and then each of their k friends.

- Bidirectional breadth-first search: We go through 2k nodes: each of S's k friends and each of D's k friends.

Of course, 2k is much less than k+k*k.

Generalizing this to a path of length q, we have this:

- BFS: $O(k^q)$
- Bidirectional BFS: $O(k^{q/2} + k^{q/2})$, which is just $O(k^{q/2})$

If you imagine a path like `A->B->C->D->E` where each person has 100 friends, this is a big difference. BFS will require looking at 100 million (100^4) nodes. A bidirectional BFS will require looking at only 20,000 nodes (2×100^2).

A bidirectional BFS will generally be faster than the traditional BFS. However, it requires actually having access to both the source node and the destination nodes, which is not always the case.

Step 2: Handle the Millions of Users

When we deal with a service the size of LinkedIn or Facebook, we cannot possibly keep all of our data on one machine. That means that our simple `Person` data structure from above doesn't quite work—our friends may not live on the same machine as we do. Instead, we can replace our list of friends with a list of their IDs, and traverse as follows:

1. For each friend ID: `int machine_index = getMachineIDForUser(personID);`
2. Go to machine `#machine_index`
3. On that machine, do: `Person friend = getPersonWithID(person_id);`

The code below outlines this process. We've defined a class `Server`, which holds a list of all the machines, and a class `Machine`, which represents a single machine. Both classes have hash tables to efficiently lookup data.

```
class Server {
  HashMap<Integer, Machine> machines = new HashMap<Integer, Machine>();
  HashMap<Integer, Integer> personToMachineMap = new HashMap<Integer, Integer>();

  public Machine getMachineWithId(int machineID) {
     return machines.get(machineID);
  }

  public int getMachineIDForUser(int personID) {
     Integer machineID = personToMachineMap.get(personID);
     return machineID == null ? -1 : machineID;
  }

  public Person getPersonWithID(int personID) {
     Integer machineID = personToMachineMap.get(personID);
     if (machineID == null) return null;

     Machine machine = getMachineWithId(machineID);
     if (machine == null) return null;

     return machine.getPersonWithID(personID);
  }
}

class Person {
    private ArrayList<Integer> friends = new ArrayList<Integer>();
    private int personID;
    private String info;

    public Person(int id) { this.personID = id; }
    public String getInfo() { return info; }
    public void setInfo(String info) { this.info = info; }
    public ArrayList<Integer> getFriends() { return friends; }
    public int getID() { return personID; }
    public void addFriend(int id) { friends.add(id); }
}
```

There are more optimizations and follow-up questions here than we could possibly discuss, but here are just a few possibilities.

Optimization: Reduce machine jumps

Jumping from one machine to another is expensive. Instead of randomly jumping from machine to machine with each friend, try to batch these jumps—e.g., if five of my friends live on one machine, I should look them up all at once.

Optimization: Smart division of people and machines

People are much more likely to be friends with people who live in the same country as they do. Rather than randomly dividing people across machines, try to divide them by country, city, state, and so on. This will reduce the number of jumps.

Question: Breadth-first search usually requires "marking" a node as visited. How do you do that in this case?

Usually, in BFS, we mark a node as visited by setting a `visited` flag in its node class. Here, we don't want to do that. There could be multiple searches going on at the same time, so it's a bad idea to just edit our data.

Instead, we could mimic the marking of nodes with a hash table to look up a node id and determine whether it's been visited.

Other Follow-Up Questions:

- In the real world, servers fail. How does this affect you?
- How could you take advantage of caching?
- Do you search until the end of the graph (infinite)? How do you decide when to give up?
- In real life, some people have more friends of friends than others, and are therefore more likely to make a path between you and someone else. How could you use this data to pick where to start traversing?

These are just a few of the follow-up questions you or the interviewer could raise. There are many others.

9.3 **Web Crawler:** If you were designing a web crawler, how would you avoid getting into infinite loops?

pg 145

SOLUTION

The first thing to ask ourselves in this problem is how an infinite loop might occur. The simplest answer is that, if we picture the web as a graph of links, an infinite loop will occur when a cycle occurs.

To prevent infinite loops, we just need to detect cycles. One way to do this is to create a hash table where we set `hash[v]` to `true` after we visit page `v`.

We can crawl the web using breadth-first search. Each time we visit a page, we gather all its links and insert them at the end of a queue. If we've already visited a page, we ignore it.

This is great—but what does it mean to visit page `v`? Is page `v` defined based on its content or its URL?

If it's defined based on its URL, we must recognize that URL parameters might indicate a completely different page. For example, the page `www.careercup.com/page?pid=microsoft-interview-questions` is totally different from the page `www.careercup.com/page?pid=google-interview-questions`. But, we can also append URL parameters arbitrarily to any URL without truly changing the page, provided it's not a parameter that the web application recognizes and handles. The page `www.careercup.com?foobar=hello` is the same as `www.careercup.com`.

"Okay, then," you might say, "let's define it based on its content." That sounds good too, at first, but it also doesn't quite work. Suppose I have some randomly generated content on the careercup.com home page. Is it a different page each time you visit it? Not really.

The reality is that there is probably no perfect way to define a "different" page, and this is where this problem gets tricky.

One way to tackle this is to have some sort of estimation for degree of similarity. If, based on the content and the URL, a page is deemed to be sufficiently similar to other pages, we *deprioritize* crawling its children. For each page, we would come up with some sort of signature based on snippets of the content and the page's URL.

Let's see how this would work.

We have a database which stores a list of items we need to crawl. On each iteration, we select the highest priority page to crawl. We then do the following:

1. Open up the page and create a signature of the page based on specific subsections of the page and its URL.
2. Query the database to see whether anything with this signature has been crawled recently.
3. If something with this signature has been recently crawled, insert this page back into the database at a low priority.
4. If not, crawl the page and insert its links into the database.

Under the above implementation, we never "complete" crawling the web, but we will avoid getting stuck in a loop of pages. If we want to allow for the possibility of "finishing" crawling the web (which would clearly happen only if the "web" were actually a smaller system, like an intranet), then we can set a minimum priority that a page must have to be crawled.

This is just one, simplistic solution, and there are many others that are equally valid. A problem like this will more likely resemble a conversation with your interviewer which could take any number of paths. In fact, the discussion of this problem could have taken the path of the very next problem.