~~5 qual questions
~~ifer Widom CS

Question 1. Suppose you are given a file of student records, where each record has three fields: ID, course, and grade. You expect to answer many questions of the form: give me all records for ID = x, where x is some constant. Describe the different types of extra mechanisms or structures that could be used so that these types of questions can be answered very efficiently. What are the trade-offs in the different mechanisms/approaches?

Answer 1. (1) Maintain a permanent hash table that maps a record ID to a list of pointers identifying where the records for that ID are located in the file. (2) Use a permanent B-tree indexing structure that, for a given ID, finds the records in the file. (3) Keep the records sorted, either as an optimization to or instead of (1) or (2). If (3) is used instead of (1) or (2), some kind of binary search would be needed. In contrasting (1) and (2), (1) can provide constant time access while (2) requires logarithmic time. However, the performance of (1) can degrade when many new records are added, while the performance of (2) should not.

Question 2: Suppose in a distributed system there is a table R(A,B) at site 1 and a table S(B,C) at site 2. A user at site 1 wishes to get the "join" of tables R and S, i.e., the user wants one record (A,B,C) for every record (A,B) in R and every record (B,C) in S such that the B values match. Assume that the most expensive operation is sending data across the network between sites 1 and 2. Describe two different algorithms for computing the join, and explain in which scenarios ~~th algorithm is preferable.

Answer 2: Algorithm (1) = all of table S is sent from site 2 to site 1; the join operation is performed at site 1. Algorithm (2) = the B values in R are sent from site 1 to site 2; the matching (B,C)s from S are sent from site 2 to site 1; the join is performed at site 1 using the S values received. Algorithm (1) is preferable in the case where most (B,C) values in S are matched in R, since it avoids the extra communication step in which R's B values are transmitted. However, if many S values are not matched in R, and the number of different B values in R is not vastly larger than S, then (2) is preferable since the extra S values are never shipped.

X-Authentication-Warning: Manta.Stanford.EDU: widom owned process doing -bs
To: Diane Shankle <shankle@ee.stanford.edu>
Subject: Re: Quals Questions
Date: Fri, 15 Jan 1999 17:04:56 -0800
From: Jennifer Widom <widom@DB.Stanford.EDU>

CS

Jennifer Widom
1999 EE Quals Question

Question for students with no database implementation background
---

Consider two tables of information stored in a computer, for example:

Employee table:

ID	name	deptNum
123	Joe	55
456	Mary	22
789	Fred	55
135	Susan	13
246	John	22
...	...	...

Department table:

num	name
10	research
55	support
22	sales
18	HR
13	develop.
...	...

Your goal is to write an algorithm that computes the "join" of these two tables based on Employee.deptNum = Department.num:

ID	emp-name	deptNum/num	dept-name
123	Joe	55	support
456	Mary	22	sales
789	Fred	55	support
135	Susan	13	develop.
246	John	22	sales
...	...	...	...

```
Suggest up to three different algorithms for computing the join of two
tables T1 and T2.  Contrast the algorithms in terms of their time
complexity and storage requirements.

POSSIBLE ANSWERS:

Algorithm 1: simple nested-loop join

for each row R1 in table T1:
  for each row R2 in table T2:
    if R1 and R2 satisfy the joining condition then combine
    R1 and R2 and append to the result table

Time complexity: O(|T1|*|T2|)
Storage requirement: essentially none

Algorithm 2: single-sort join

sort table T1 on the joining value;
for each row R2 in T2:
  use binary search on sorted T1 to find all matching values and
  add to the result table

Time complexity: O(|T1|*log(|T1|)) to sort T1
                 O(|T2|*log(|T1|)) for second step

Algorithm 3: sort-merge join    Sort both

sort table T1 on the joining value;
sort table T2 on the joining value;
traverse the two tables linearly (with some "backtracking" for
  duplicate values), matching join values and adding joining tuples
  to the result table

Time complexity: O(|T1|*log(|T1|)) to sort T1
                 O(|T2|*log(|T2|)) to sort T2
                 O(max(|T1|,|T2|) for "merge" phase assuming not too
                   many duplicate joining values
Storage requirement: not much, depends on sorting algorithm used

Algorithm : hash join

set up hash table;
for each row R1 in T1:
  hash R1's join value and put R1 in the appropriate hash bucket
for each row R2 in T2:
  hash R2's join value;
  find all matching tuples in the hash bucket and add to the result table

Time complexity: |T1|+|T2| assuming well-distributed hash table
Storage requirement: O(|T1|) for hash table
```

?

Question for students with database implementation background
----

Consider the standard tuple-based nested-loop join algorithm for computing T1 JOIN T2:

```
for each row R1 in T1:
  for each row R2 in T2:
    if R1,R2 satisfy the join condition then add R1/R2 to result
```

Suggest three separate possible improvements to this algorithm. Assume a standard DBMS storage system and query processing context. Improvements could depend on additional assumptions or scenarios.

ANSWER:

1. Nested-block join

Process T1 and T2 block-at-a-time instead of row-at-a-time. Considerably reduces the number of times T2 is scanned, without incurring extra I/O's for T1. ☆ "block at a time"

2. "Rocking"

For T2, scan it forwards the first time, backwards the second time, forwards the third time, etc. Takes advantage of LRU page replacement policy typically used by database buffer managers. forward - backward

last recent use

3. Use of keys

If the join condition is T1.A = T2.B and B is a key for T2, then once a match is found the algorithm can break out of the inner loop.

4. Use of index

If the join condition is T1.A = T2.B and there is an index on T2.B then the inner loop can find matching T2 rows using the index instead of by scanning the whole relation. (Also works for inequality join conditions if the index is a B-tree.)

```
To: Diane Shankle <shankle@ee.stanford.edu>
Subject: Re: Quals Meeting Today!
Date: Mon, 29 Jan 2001 20:43:32 -0800
From: Jennifer Widom <widom@DB.Stanford.EDU>

Jennifer Widom 2001 EE quals questions with sample solutions:

======================================================================

Consider a binary tree with values in each node.  That is, each node N
of the tree has:

  N.value: an integer
  N.left: the root of the left subtree, or NULL
  N.right: the root of the right subtree, or NULL

Every node has either two children or zero (i.e., N.left = NULL iff
N.right = NULL), but trees need not be balanced.

======================================================================

(1) Write a recursive function Sum(T) that returns the sum of all
values in the binary tree rooted at T.  Do not use any global
variables.

  Sum(T):
    if T.left = NULL then return(T.value)
    else return(T.value + Sum(T.left) + Sum(T.right))

======================================================================

(2) Write a recursive function Height(T) that returns the length of
the longest path from the root of the binary tree rooted at T to a
leaf.  Do not use any global variables.

  Height(T):
    if T.left = NULL then return(0)
    else return(1 + max(Height(T.left), Height(T.right)))

======================================================================

(3) Write a recursive function MinTwo(T) that returns the two smallest
values in the binary tree rooted at T.  You may assume the tree
contains at least 2 (therefore 3) nodes, and that each value in the
tree is unique.  Do not use any global variables.

  MinTwo(T):
    // local variable temp has type set of integers
    if T.left = NULL then return({T.value})
    else begin
       temp := MinTwo(T.left) UNION MinTwo(T.right) UNION {T.value};
       return({min(temp), min(temp - min(temp))})
    end

======================================================================
```

To: shankle@ee.Stanford.EDU
Subject: EE quals question/solution
Date: Fri, 25 Jan 2002 17:54:18 -0800
From: Jennifer Widom <widom@DB.Stanford.EDU>

Diane,

I suddenly realized I forgot to turn in my EE Quals question/ solution. I can't remember if you're the one to take it, but if not please forward accordingly.

Thanks,
Jennifer Widom

===========================

2002 EE Quals, Prof. Jennifer Widom

Problem
=======

Consider directed graphs with a single source (root) node R from which there is at least one path to every other node N in the graph.

We can represent such graphs in a couple of ways:

(1) As a data structure of nodes and edges. Each node N has i >= 0 out-neighbors (children) accessed as N.1, N.2, ..., N.i
[show example]

(2) As a K x K square matrix M for a graph with K nodes. M[x,y] = 1 if there is an edge from node x to node y; M[x,y] = 0 otherwise
[show same example]

Write a program that determines whether such a graph contains a cycle. The program should return YES if there is one or more cycles, NO otherwise.

* You may use whichever of the two graph representations you prefer.

* You may use any pseudocode notation you like, including function definitions and calls if it helps you.

* Your solution will be graded on simplicity as well as correctness.

Solution
========

Here are two possible solutions but there are many correct variants.

Using representation (1):

```
  main program: cycle(R, {R})

  function cycle(N, seen-set)
    if N has no children return(NO)
    else if any of N.1, N.2, ..., N.i are in seen-set return(YES)
    else if cycle(N.i, seen-set U {N.i})=YES for any i
      then return(YES) else return(NO)
```

Using representation (2):

```
  repeat until M is unchanged:
    M <- M + M*M
  if M[x,x] > 0 for any x then return(YES) else return(NO)
```

Problem
=======

Jennifer Widow

Towers of Hanoi

There are three posts, P_1, P_2, and P_3. Post P_1 starts with a tower of N disks on it, D_1, D_2, ..., D_N, of strictly decreasing size from bottom to top. The goal is to move all N disks from post P_1 to post P_2, possibly via post P_3, subject to:

(1) At most one disk may be moved at a time.
(2) No disk may ever be placed on top of a smaller disk.

Specifically, write a general procedure:

Move({disk1, disk2, ..., diskM}, post1, post2, post3)

that emits a sequence of instructions for moving disk1, disk2, ..., diskM from post1 to post2, possibly via post3. To solve the original problem we call:

Move({D_1, D_2, ..., D_N}, P_1, P_2, P_3)

Each emitted instruction is of the form "move D_i from P_j to P_k".

Hint on request:

Use recursion-- note that Move() can be called with any set of disks and any parameter ordering of the three posts.

Additional/alternate problems:

(A1) Write a function that takes an argument N and returns the number of moves required to solve the problem with N disks.

(A2) What is the computational complexity of the problem (in #disks)?

Solution

```
Move({disk1, disk2, ..., diskM}, post1, post2, post3):
  If M=1 then emit "move [disk1] from [post1] to [post2]"
  Else
    Move({disk2, disk3, ..., diskM}, post1, post3, post2)
    Move({disk1}, post1, post2, post3)
    Move({disk2, disk3, ..., diskM}, post3, post2, post1)
```

(A1) f(1) = 1; f(N > 1) = 2*f(N-1) + 1

(A2) Exponential in N: O(2^N) Specifically, f(N) = 2^N - 1

Date: Wed, 01 Mar 2006 09:34:51 -0800
From: Jennifer Widom <widom@cs.stanford.edu>
X-Accept-Language: en-us, en
To: Diane Shankle <shankle@ee.Stanford.EDU>
Subject: Re: Reminder Quals Question 2006

Jennifer Widom EE Quals 2006

QUESTION
========
Consider a hierarchical sensor data processing setup with:

(1) A high-end processor H at the root.

(2) A set of k low-end processors L_1, L_2, ..., L_k that can send values to H.

(3) For each processor L_i, a set of n_i sensors that can send values to L_i.

Each sensor reads one value and sends it to its parent L_i. At the root H, we want to know the average of all the sensor values.

Let a_L denote the cost of performing a binary arithmetic operation (e.g., addition, division) at an L_i processor, and let a_H denote the same for processor H. We expect a_H to be lower than a_L.

Let m_i denote the cost of sending a message with a single numeric value (and a few status bits if needed) from L_i to H. Assume this cost metric is compatible with the one used for a_L and a_H.

** Describe alternative algorithms for performing the average
** computation, and explain how to decide which of your algorithms is
** cheapest.

EXTRA 1: Modify your answer for the case where we want to compute the minimum instead of the average. Assume comparison costs are the same as arithmetic costs: a_L and a_H for a binary compare.

EXTRA 2: Modify your original answer for the case where we want to compute the median instead of the average.

ANSWER
======
Each L_i has to decide whether to:

(a) Simply pass its sensor values on to H, or
(b) Compute a sum and count of its sensor values and pass those to H.

In either case, H must compute the sum of all the values it receives and divide it by the total counts.

Cost of option (a):

Arithmetic at L_i: 0
Messaging: n_i * m_i
Arithmetic at H due to L_i: a_H * n_i

Cost of option (b):

Arithmetic at L_i: a_L * (n_i - 1)

```
   Messaging: 2 * m_i
   Arithmetic at H due to L_i: 2 * a_H

Prefer option (a) when (n_i * m_i) + (a_H * n_i) <
                       (2   * m_i) + (a_L * (n_i - 1)) + (2 * a_H)

EXTRA 1: Very similar to original except replace:
  (a_H * n_i) with (a_H * (n_i - 1))
  (2 * m_i) with m_i
  (2 * a_H) with a_H

EXTRA 2: Except for certain extreme cases, all values must be
transmitted to a single site in order to compute a median, so only
option (a) is feasible.
```