95/100

6.830 **Dan Ports**

Database Systems drkp@mit.edu

2007/10/25 Collaborators: {amdragon, iyzhang}

Problem Set 3

Problem 1:

9/10

We'll assume that, within each histogram bucket, the employees are evenly distributed. We'll further assume that, for this problem, age and salary are entirely independent.

a)

$$\begin{aligned}\mathbf{Pr}[\texttt{sal} \leq 40,000] &= \mathbf{Pr}[\texttt{sal} < 20,000] + \mathbf{Pr}[\texttt{sal} \in [20,000, 35,000)] + \frac{5000}{15000}\mathbf{Pr}[\texttt{sal} \in [35,000, 50,000)] \\ &= 0.05 + 0.15 + \frac{1}{3}0.25 \approx 0.283\end{aligned}$$ ✓

b)

$$\mathbf{Pr}[\texttt{age} = 43] = \frac{1}{10}\mathbf{Pr}[\texttt{age} \in [40-49]] = 0.035$$ ✓

c)

$$\begin{aligned}\mathbf{Pr}[\texttt{sal} < 50,000 \cup \texttt{age} \geq 50] &= \mathbf{Pr}[\texttt{sal} < 50,000] + \mathbf{Pr}[\texttt{age} \geq 50] - \mathbf{Pr}[\texttt{sal} < 50,000]\mathbf{Pr}[\texttt{age} \geq 50] \\ &= (0.05 + 0.15 + 0.25) + (0.25 + 0.1) - (0.45)(0.35) \\ &= 0.45 + 0.35 - 0.1575 = 0.6425\end{aligned}$$ ✓

d) We'll consider this piecewise by age.

This is correct, but better to state why!

$$\begin{aligned}\mathbf{Pr}[\texttt{sal} > 500 \times \texttt{age} \mid \texttt{age} \in [20,29]] &= 0.05\frac{20000-12500}{20000} + 0.95 &&= 0.96875 \\ \mathbf{Pr}[\texttt{sal} > 500 \times \texttt{age} \mid \texttt{age} \in [30,39]] &= 0.05\frac{20000-17500}{20000} + 0.95 &&= 0.95625 \\ \mathbf{Pr}[\texttt{sal} > 500 \times \texttt{age} \mid \texttt{age} \in [40,49]] &= 0.15\frac{35000-22500}{15000} + 0.8 &&= 0.925 \\ \mathbf{Pr}[\texttt{sal} > 500 \times \texttt{age} \mid \texttt{age} \in [50,59]] &= 0.15\frac{35000-27500}{15000} + 0.8 &&= 0.875 \\ \mathbf{Pr}[\texttt{sal} > 500 \times \texttt{age} \mid \texttt{age} \in [60,69]] &= 0.15\frac{35000-32500}{15000} + 0.8 &&= 0.825\end{aligned}$$

Having calculated these conditional probabilities, we can multiply each of them by the probability of their corresponding age bucket and sum:

$$\mathbf{Pr}[\texttt{sal} > 500 \times \texttt{age}] = 0.1{\times}0.96875{+}0.2{\times}0.95625{+}0.35{\times}0.925{+}0.25{\times}0.875{+}0.1{*}0.825 = 0.913125$$

Problem 2:

c) We can see that the sets of employees with salary less than 50,000 and employees over 50 are disjoint: employees over 50 make up 0.35 of the population, and so they must earn salaries in the upper 0.35, *i.e.* over 50K. So:

$$\begin{aligned}\mathbf{Pr}\left[\texttt{sal} < 50,000 \cup \texttt{age} \geq 50\right] &= \mathbf{Pr}\left[\texttt{sal} < 50,000\right] + \mathbf{Pr}\left[\texttt{age} \geq 50\right] \\ &= (0.05 + 0.15 + 0.25) + (0.25 + 0.1) \\ &= 0.45 + 0.35 - 0.8\end{aligned}$$

d) Employees aged over 30 are in the top 0.9 by age, so they must be in the upper 0.9 by salary as well. This means they must earn at least \$20K, and therefore they all satisfy the predicate. So we need to consider only the lowest end of the 20-29 range. Again, we'll assume a uniform distribution within each bucket, and that salaries start at \$0. Then, in this region the inverse cumulative distribution function of salary is given by the function $f(x) = 400000x$, and the inverse cumulative distribution function of age is given by $g(x) = 20 + 100x$. The point at which the salary exceeds 500 times the age is the value of x for which $500g(x) = f(x)$:

$$\begin{aligned}500\,(20 + 100x) &= 400000x \\ 10000 + 50000x &= 400000x \\ x &= \frac{1}{35}\end{aligned}$$

So the selectivity is $1 - \frac{1}{35} = \frac{34}{35} \approx 0.9714$.

Problem 3:

a) Serial: yes. The equivalent serial ordering is T2 followed by T1.
Locking: Would not occur. T2 would acquire a shared lock on A at step 3, preventing T1 from obtaining the exclusive lock at step 4.
Optimistic: yes. When T1 commits, its read set does not intersect T2's write set.

b) Serial: yes. Again, the equivalent serial ordering is T2 before T1.
Locking: yes. T2 drops its locks on commit, so T1 is able to gain exclusive locks after.
Optimistic: yes. T1's read set does not intersect T1's write set at the time T1 commits.

c) Serial: yes. T2, T1, T3 is a valid serial ordering
Locking: no. T2 acquires an exclusive lock on B at step 2, which would prevent T3 from acquiring a shared lock at step 3.
Optimistic: no. It would fail when T1 commits, since its read set contains B, which is also in T2's write set.

d) Serial: no. T3 reads A before T1 writes A, but T3 writes B after T1 reads B.
Locking: no. 2-phase locking generates only serializable orderings, which this is not. Specifically, step 3 could not happen, because T2 would need to hold an exclusive lock on B to write it in step 1, preventing T3 from later acquiring a shared lock.
Optimistic: no. Optimistic concurrency control would not generate an non-serializable ordering like this one. Specifically, it would fail when T1 commits, since its read set contains B, which is also in T2's write set.

e) Serial: yes. The ordering is T1, T2. However, the transactions are not commit-ordered since T2 commits before T1.
Locking: no. T1 acquires an exclusive lock on A in step 1 and does not release it until step 6, so T2 could not acquire an exclusive lock in step 2.
Optimistic: no. When T1 commits, its read set intersects the write set of T2.

f) Serial: yes. The ordering is T1, then T2.
Locking: no. T1 acquires an exclusive lock on A in step 1 and does not release it until step 4, so T2 could not acquire an exclusive lock in step 3.
Optimistic: yes. When T2 commits, its write set is empty, so it cannot conflict with T1.

Problem 4:

a) First, note that each tuple in `books` is 112 bytes, so 8 tuples fit in a page, and there are 125 data pages in the table. There would then need to be three index pages (one top-level and two second-level).

For this query, since there is no index on `pubyear`, the only reasonable query plan is to perform a sequential scan over `books`. Clearly this requires reading all data pages; we'll conservatively assume that it also requires accessing all index pages, though a clever B^+-tree implementation might be able to get away with less. Thus, 128 shared locks would need to be acquired.

b) Each tuple in `authors` is 204 bytes, so 4 tuples fit on each page; there are 500 data pages. Following the same reasoning as with `books`, there are six index pages. For `bookauths`, each tuple is 8 bytes, so 125 fit on each page; the table has 32 data pages and one index page.

For this query, the best query plan is a left-deep tree that begins with a sequential scan to filter `books`, joins it with `bookauth` using an index scan over `bookauth.bookid`, and joins the result on `bookauth.authorid` = `authors.id` using the index on `authors.id`.

Sequential-scanning `books` requires acquiring shared locks over all of its pages, as before, so 128 S locks. We'll assume that only one book matches (and furthermore that this particular book has two authors). Finding the matching entries in `bookauths` requires reading the index page, then reading the matching data page. This requires acquiring two shared locks.

Finally, we need to perform an index lookup on `authors` for both of the resulting `authorids`. This requires accessing the top-level index page, up to two different second-level index pages, and up to two different data pages, so 5 shared locks need to be acquired.

The total number of shared locks required is $128 + 2 + 5 = 135$; no exclusive locks are required.

c) The query plan for this query must involve sequential-scanning `authors`, updating any matching tuples, since there are no useful indexes. This requires acquiring shared locks on all 500 data pages, and conservatively (following our earlier assumption) all 6 index pages. We'll also need to acquire exclusive locks on any pages containing tuples that are changed. Lacking any information about the distribution of addresses of authors, this could be anywhere from 0 to 500 pages. No index pages, however, will need to be modified. So this query must acquire 506 shared locks, and may need to upgrade up to 500 of them to exclusive mode.

Problem 5:

LSN	Type	TransID	PrevLSN	PageID	Data
1	Begin	1			
2	Update	1	1	1	page 1 before/after UPDATE
3	Update	1	2	2	page 2 before/after UPDATE
4	Update	1	3	3	page 3 before/after INSERT
5	End	1	4		

10/10

Problem 6:

a) T3 and T4 were running at the time of the crash.

b) T3 and T4 will therefore be undone.

c) All data pages were flushed at the time of the last checkpoint, so the only page that was dirty at the time of the crash was X, for transaction T4. This update will be redone during ARIES's REDO phase.

d) The updates corresponding to transactions T3 and T4 will be undone during the UNDO phase. These are the updates to Z, W, and X.

10/10

Problem 7:

The obvious use for NVRAM would be to persistently store the log tail in order to reduce the amount of time the database spends waiting for synchronous disk writes, but it's not clear that one page would be enough to be effective for this task: each log record could be more than a page long, since it needs to include physical undo/redo information for the page it modified.

Instead, we can use the NVRAM to store the transaction table and dirty page table. These are likely to fit in a single page, and storing them persistently can give a large speed improvement for recovery, in addition to a small improvement during normal database operation.

With the transaction table and dirty page table in NVRAM, there is no need to perform checkpoints to disk during normal operation. There is also no need for the analysis phase of recovery, since that phase exists solely to recover the state of these two structures. Finally, for each operation that needs to be REDOne, there is no need to check on disk whether the page was flushed after the last checkpoint, since the dirty page table in the NVRAM will always be up to date; this means that deciding whether an operation needs to be redone can be done using only the dirty page table and log entries.

good!