CS632 Final Exam Part B

A. Demers

6 May 2003

I hope to add a short query optimization question sometime tonight. Note an additional part has been added to Problem 1.

As usual, you may consult others for ideas and proof approaches, but please *reference your sources* and write up answers independently.

1 The Oracle(tm) Isolation Rules

Here is a somewhat simplified description of the isolation rules used by Oracle.

The system maintains an event counter called the System Change Number (SCN), which is effectively a clock. For any entity x and any SCN (or time) t, the system can return the last value written to x by a transaction that committed before t. Thus, for any time t a transaction can arrange to get a *consistent snapshot* of the database, consisting of the values written by those transactions that committed before t.

Dirty reads do not happen. Each entity x is tagged with the transaction that last wrote it. If that transaction has not (yet) committed, the value of the entity is never returned to a reader, but instead data from the *rollback segment* (which is essentially an undo log) is used to find and return the appropriate value written by a committed transaction.

The operations are supporting this model are

R(x, t): Return the value written to entity x by the last transaction to commit before t.

R(x): Equivalent to R(x, C) where C is the current SCN.

L(x): Acquire an exclusive lock on entity x.

U(x): Release the lock on entity x.

W(x): Write entity x.

C: Commit the transaction.

A: abort the transaction.

As usual, we shall assume a transaction is not allowed to re-read an entity that it has already written.

Problem 1a: Sketch how an ARIES-like recovery algorithm can support the operations described above. Obviously, the interesting operation is R(x, t). Don't worry about fine-granularity locking – you may assume a locked entity occupies exactly one page. But don't oversimplify either – your algorithm should support a transaction that modifies more entities than will fit in the buffer pool, and committing a transaction should require forcing the log but not waiting for dirty pages to be flushed from the buffer pool. Discuss checkpointing and archiving/truncating the log.

Problem 1b: By default, the locking protocol used by Oracle is strict 2-phase locking on writes (using exclusive locks) and *no locking at all* on reads. Each read is done at the current SCN (that is, transactions use R(x) rather than R(x, t) in the above description).

Show by example that the default Oracle locking protocol can generate nonserializable schedules.

(In practice, a transaction consists of multiple SQL queries, each of which accesses many entities. All the read access of a single SQL query are performed as of the same SCN, so that each SQL query sees a consistent snapshot. However, different SQL queries in the same transaction are executed at different SCNs and thus see different snapshots. Feel free to ignore this detail, and think of each SQL query as a single read.)

Problem 1c: An alternative rule does strict 2-phase locking on writes as above, but in addition associates with each transaction A a "snapshot timestamp" t_A, which is the SCN when the transaction was started. Every read in A is performed at time t_A – that is, each read R(x) is replaced by R(x, t_A). In addition, the following test is performed for each lock operation L$_A(x)$:

> After the lock has been granted, if the commit time of the last transaction that wrote x is greater than t_A, then abort A.

Show that this protocol can also generate nonserializable schedules.

Problem 1d: (We are trying desperately to find some way to guarantee serializable schedules in Oracle.) Consider *static* 2-phase locking, that is, strict 2-phase locking with the additional requirement that all locks must be acquired at the beginning of the transaction, before the first read or write operation. Of course, static 2-phase locking is not realistic – it is possible to predict the complete set of locks that will be requested only for extremely simple transactions. Nevertheless . . . Let the snapshot timestamp of a transaction be the SCN after all locks have been acquired, and perform all reads as of that SCN as in (1c). Using this rule, are nonserializable schedules possible? Give an example, or prove that no such example exists.

Under rule (1d), what happens to transactions / schedules that caused transaction aborts under rule (1c)?

Problem 1e: On the other hand, consider the following class of transactions, parameterized by w, a worker:

Compute a, the average salary of all workers;

Update the salary of w by $s \leftarrow s + \epsilon * (s - a)$

You may choose the magnitude (and sign) of ϵ according to your own political leanings. Each execution of a transaction of this form does a large number of reads at the same snapshot time to compute the average (this takes a fair amount of real time) and does a single write at the end to update the worker's salary.

Now suppose three instances of this transaction are started simultaneously, for workers Steve, Larry and Bill. What is the probability of a deadlock using the Oracle default rules (1b)? What about using strict 2-phase locking with shared locks for reads and exclusive locks for writes? What about strict 2-phase locking with exclusive locks for all accesses? Discuss the advantages and disadvantages of these schemes. A detailed discussion of snapshot isolation can be found at

http://www.cs.umb.edu/ isotest/snaptest/snaptest.pdf

but should not be required to answer this question.

2 NULL Values

Recall the notion of strong and weak satisfaction of a FD f using the substitution principle for NULL values. That is, on relation $\boldsymbol{r}$ (which may contain NULLs) the value of FD f is *true* if all possible substitutions of values for the NULLs in $\boldsymbol{r}$ generate relations that satisfy f;

false if all possible substitutions of values for the NULLs in $\boldsymbol{r}$ generate relations that do not satisfy f;

unknown if some substitutions generate relations that satisfy f and some substitutions generate relations that do not satisfy f.

Then f is said to be

strongly satisfied by $\boldsymbol{r}$ if its value is *true*,

violated by $\boldsymbol{r}$ if its value is *false*, and

weakly satisfied by $\boldsymbol{r}$ if its value is either *true* or *unknown*.

Recall also the set $\{FD_1, FD_2, FD_3\}$ of inference rules for Functional Dependencies:

$$\frac{}{X \to X}$$

$$\frac{X \to YZ}{X \to Y}$$

$$\frac{X \to YZ \qquad Z \to W}{X \to YZW}$$

which are sound and complete for ordinary satisfaction of functional dependencies.

Problem 2a: Show (by example) that the set $\{FD_1, FD_2\}$ is sound for weak satisfaction, but FD_3 is unsound.

Problem 2b: In lecture we stated without proof that FD_3 could be replaced by the "extended union" rule FD_4:

$$\frac{XZ \to Y \qquad X \to W}{XZ \to YW}$$

to produce a sound and complete system for weak satisfaction. Prove this.

In the context of weak satisfaction, give a revised algorithm to compute the closure of a set of attributes with respect to a set of FDs.

3 Hypergraph Protocol

Recall a locking protocol π is a mapping that assigns to each prefix B of a locked transaction a set $\pi(B)$ of entities; the interpretation is that if $x \in \pi(B)$ then $L(x)$ could be allowed as the next step of this transaction.

Given a set τ of transactions closed under truncation define the hypergraph $H(\tau)$ which has the entities of the database as its vertices, and which has as its hyperedges all sets h of entities such that there is a transaction in τ that accesses exactly the entities in h.

We define the locking protocol π_τ in which lock steps come immediately before the first access of the corresponding entity, and unlock steps come as early as possible according to the following rule:

> A previously accessed entity x can be unlocked if the set of other entities that remain locked separates x in $H(\tau)$ from any entity that is locked later in the transaction.

Problem 3a: (This is Problem 6.18 from Papadimitriou). Show the above rule is actually a locking protocol – that is, show how to state it in terms of which *locking* steps are allowed rather than which *unlocking* steps are allowed.

Problem 3b: The above construction produces locking protocol that is safe. Is it (or can you enhance it to be) deadlock-free as well? Justify your answer.