

Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.893 Database Systems: Fall 2004

Quiz I Solutions

There are 14 questions and 16 pages in this quiz booklet. To receive credit for a question, answer it according to the instructions given. *You can receive partial credit on questions if you circle some of the correct answers, but we may subtract points for circling a wrong answer.* You have **80 minutes** to answer the questions.

Write your name on this cover sheet AND at the bottom of each page of this booklet.

Some questions may be harder than others. Attack them in the order that allows you to make the most progress. If you find a question ambiguous, be sure to write down any assumptions you make. Be neat. If we can't understand your answer, we can't give you credit!

**THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.
NO PHONES, NO LAPTOPS, NO PDAS, ETC.**

Median: 67.5
25th percentile: 58
75th percental: 70
Mean: 63.9

Name: Solutions

I Reading Questions

1. [6 points]: Dana Bass is building a database system, and has implemented both a Hybrid Hash Join and a Sort-Merge join (as described in the “Join Processing” paper by Shapiro et al.) She is programming the optimizer of her database system with a set of rules to decide when to use hash joins and when to use sort-merge joins. Under what circumstances would hash join definitely be preferred, assuming that an equi-join (e.g., of the form “ $S.a = R.b$ ”) is being performed? Suppose that the join is between two relations, S and R , and that $|S|$ denotes the size of relation S , and that $|R| \leq |S|$.

(Circle ALL that apply)

A. Always.

Not true, since B is true.

B. If S or R is unsorted and $|S|$ is much larger than $|R|$.

This is true. Sort-merge join’s performance is inferior to hash join due to the overhead of sorting, so if S or R is unsorted, it’s likely to perform worse. However, if $|S|$ is about the same size as $|R|$, hash join may have higher I/O costs than sort-merge due to the space overhead introduced by hashing.

C. Whenever there are fewer than $\sqrt{|S|}$ pages of free memory.

It’s not clear which approach is preferred in this case. The fact that there is little available memory relative to $|S|$ means that either approach is likely to have to write more runs, but the I/O costs should be comparable in either case.

D. If both S and R fit in main memory.

If they both fit into main memory, the cost of sorting will likely dominate the cost of hashing, so this may be true, although other tradeoffs (such as the increased space overhead of hashing) make this tradeoff somewhat unclear. No points were awarded either way for this answer.

2. [6 points]: The purpose of schema normalization is to :

(Circle ALL that apply)

A. Eliminate redundant data stored in the database.

This is correct. A normalized database has fewer repeated values – redundancy is replaced with foreign-key references to the redundant data.

B. Reduce the number of joins required to satisfy a query.

Incorrect. Schema normalization tends to increase the number of joins required, as it creates new tables.

C. Reduce the number of anomalies that can occur during inserts, deletes, and updates.

True – this is the main purpose of schema normalization.

D. Convert the data to a canonical form to promote schema integration.

This is false. Schema normalization doesn’t have anything to do with schema integration, except that they both involve transformations of the schema.

3. [6 points]: An OODBMS (such as ObjectStore) offers some advantages over a relational database, such as:

(Circle ALL that apply)

A. An OODBMS provides a more expressive query language than SQL and other relational languages.

Generally speaking, this is not true, although no points were removed for circling it. OODBMSes can be seen as more expressive because they allow any computation to be applied to a persistent object; however, the query language itself is usually less expressive because it is difficult to support sophisticated joins.

B. An OODBMS avoids the “impedance mismatch” problem.

This is true – this is one of the stated goals of an OODBMS.

C. An OODBMS avoids the “phantom” problem.

False. Object databases have nothing to do with the phantom problem.

D. An OODBMS provides higher performance concurrency control than most relational databases.

This is false. There’s no reason to believe that the locking done in an OODBMS like ObjectStore is any more efficient than the locking done in a traditional RDBMS – in fact, it’s likely to be higher overhead because it is distributed and requires features like lock callback.

E. An OODBMS provides faster access to individual data objects once they have been read from disk.

True. Once an object has been read from disk in an OODBMS, the program can manipulate it as though it was a standard memory resident object.

4. [6 points]: Views in a database system are important because:

(Circle ALL that apply)

A. They improve the efficiency of query execution.

False. Unless they are materialized, views do nothing for performance.

B. They help provide data independence.

True. Views allow programs to be independent of the actual physical layout of the data.

C. They allow the schema to change without forcing existing applications to be recompiled.

True, for similar reasons as in B. When the schema changes, a view can be created that looks like the old schema.

D. They can be rewritten using query rewrite.

Although this is true, this is not a reason views are important. Rewrite is a technique for making views usable in a database system.

E. They help with access control by allowing users to see only a particular subset of the data in the database.

This is true.

5. [6 points]: In ARIES-style recovery, logging is “physiological” – that is the REDO phase is physical and the UNDO phase is logical. Possible reasons for this are:

(Circle ALL that apply)

- A. Physical REDO allows the REDO records to be smaller than they would be with logical REDO.
This is false. Physical records are typically larger than logical records.
- B. Logical UNDO allows the UNDO records to be smaller than they would be with physical UNDO.
This is true – smaller records are an advantage of logical logging.
- C. Logical REDO is not possible because the ARIES protocol cannot ensure that the database is in an action-consistent state after a crash.
This is true.
- D. Physical UNDO is difficult to implement because physical pages may be laid out differently during UNDO than they were when the action being undone was originally taken.
Also true. See the slides from class on 10/15.

6. [6 points]: According to the System-R* paper, the best strategy for computing a distributed join is:

(Circle ALL that apply)

- A. Always the same as the best local strategy.
Not true. Additional memory for the join in the distributed case, as well as the additional cost of transferring messages between the various sites, can cause the best local strategy to perform poorly in a distributed environment.
- B. Usually Fetch-Matches, because Ship-Whole requires more bytes to be transferred over the network.
Not true. Although Fetch-Matches does require fewer bytes to be sent, it requires many more messages, and in most network configurations, the number of messages dominates the number of bytes transferred.
- C. Usually Bloom-join, because it transfers much less data than semi-join.
Although Bloom-join is the best strategy, it's not because it transfers less data – the data requirements are about the same.
- D. Usually Bloom-join, because it requires less local processing than semi-join.
This is true.
- E. The same as the best local strategy if the database is running on a fast network (where message costs are of little importance.)
Not true – for the same reason as in A.

II The Friendinator : The new, new way to meet people

Dana Bass is starting a “social networking” website where people post information about themselves and can search for other users with similar interests and backgrounds. She has decided to call her service “The Friendinator”. To distinguish The Friendinator from other social networking services like Friendster and Orkut, she wants to focus on providing very high performance searches over her database of users, something she believes the other services have trouble with. She decides to store her data using a relational database.

Dana decides that the database used in The Friendinator will consist of four tables: an `accounts` table that contains login information and other information about the users of the system, an `interests` table that contains information about the various interests that people might have, a `likes` table that records information about the interests of all the users, and a `friends` table that keeps track of friendships between users.

She chooses to use the following schemas (only the relevant fields are shown here):

```
accounts
{
    user_id : int primary key,
    first_name : string,
    last_name : string,
    email : string,
    sex : char,
    age : int,
    home_town : string,
    status : int,
    home_page : url ...
}
```

This table keeps track of the users in the system. Most of these fields should be self explanatory. The `status` field is used to keep track of the account status (e.g., whether the user has a paid account, whether he or she has received a particular promotion, etc.)

```
interests
{
    interest_id : int primary key,
    interest : string,
    creator : int foreign key accounts.user_id,
    home_page : url ...
}
```

This table keeps track of the kinds of “interests” a user can have – for example, an interest might be “baseball”, in which case the URL would point to a page about baseball and have a list of users who were interested in it.

```
likes
{
    user_id : int foreign key accounts.user_id,
    interest_id : int foreign key interests.interest_id,
} primary key user_id, interest_id
```

Name: Solutions

This table keeps track of which interests each user has. Users may have multiple (or no) interests.

```
friends
{
    friend1 : int foreign key accounts.user_id,
    friend2 : int foreign key accounts.user_id
} primary key friend1, friend2
```

This table keeps track of which users are friends with each other. Friendship is a symmetric relationship (e.g., if Alice is friends with Bob, Bob is also friends with Alice.) Each user may have multiple (or no) friends.

The Friendinator allows users to look for other users using a set of predefined search queries. One such query allows users to find people from their home town who are about the same age and with whom they share at least one common interest. Dana writes the following query to generate a list of such users and their common interests:

```
SELECT a2.first_name, a2.last_name, i.interest
FROM accounts AS a1, accounts AS a2, likes AS l1, likes AS
l2, interests AS i
WHERE a1.user_id = uid
AND a2.user_id != uid
AND a1.home_town = a2.home_town
AND a1.age BETWEEN a2.age - 2 AND a2.age + 2
AND l1.user_id = a1.user_id
AND l2.user_id = a2.user_id
AND l1.interest_id = l2.interest_id
AND l2.interest_id = i.interest_id
```

In this query, *uid* refers to the user id of the user issuing the query.

Initially, there are no indices. There are 10,000 users, 100 possible interests, and 20,000 entries in the `likes` table. Users' ages are uniformly distributed around the range 0-100, and their home towns are uniformly selected from 100 possible locations.

To help optimize the performance of this query, Dana wants to understand how a database system would most likely execute this query. She comes to you for help.

Query Optimization Questions

7. [5 points]: Which of these expressions would the query optimizer most likely decide to execute first (e.g., as one of the bottom-most nodes in the query plan) if it wants to generate an optimal plan:

(Circle ONE.)

Typically, the best choice of bottom-most node is the one that will reduce the size of the input relation. Thus, clearly neither of the multi-table join predicates would be a good choice to run first; either of the filters over user_id would be appropriate. Since the predicate over a1 is most selective, B would be the best choice – however, due to the wording of this problem, answer C was also accepted.

A. a1.home_town = a2.home_town

B. a1.user_id = uid

C. a2.user_id != uid

D. l1.user_id = a1.user_id

8. [5 points]: Which of these expressions would result in the worst overall query performance if it was executed first?

(Circle ONE.)

A. a1.user_id = uid

This is a highly selective predicate and is not the correct choice.

B. a1.home_town = a2.home_town

Each of the 10,000 accounts in a1 will join with 1% (or 100) of the accounts in a2, yielding approximately $10,000 \times 100 = 10,000,000$ records if this is run on unfiltered versions of a1 and a2.

C. a1.age BETWEEN a2.age - 2 AND a2.age + 2

Each of the 10,000 accounts in a1 will join with 5% (or 100) of the accounts in a2, yielding approximately $10,000 \times 500 = 50,000,000$ records if this is run on unfiltered versions of a1 and a2, so this is a better choice than B.

D. l1.user_id = a1.user_id

Each of the 20,000 likes in l1 will join with exactly 1 of the accounts in a1, yielding approximately 20,000 records if this is run on unfiltered versions of l1 and a1. Thus, C is a better choice.

9. [10 points]: The database that Dana is using provides support for nested loops, sort-merge, and hash-join algorithms similar to those we have studied in class. None of the relations are initially sorted. Each disk page holds 10 tuples of any of the relations, and there are 100 pages of memory available to the database. The smallest number of I/Os that this query could require is approximately:

(Circle ONE.)

Without indices, this query requires 5 tables to be read: a1, a2, l1, l2 and i. If we execute the plan shown in Figure 1, we can use output of the scan over a2 as the outer relation of a nested loops join between the filtered a1 and a2 – since the filtered a1 is only a single record, we can keep it in memory. The output of this join will be of approximate size $10,000 \times .05 \times .01 = 5$ records, which will clearly fit into memory. We can perform the joins with l1 and l2 similarly, using l1 or l2 as the outer relation and joining them with the memory-memory resident inner relation. Since each user has about 2 likes, the result of these joins will also fit into memory. Finally, the join with interests can be performed entirely in-memory. Thus, we never need to write any intermediate data to disk to complete this join, and the total I/O cost is simply the cost to scan all of the tables, which is $1000 \times 2 + 2000 \times 2 + 10 = 6010$.

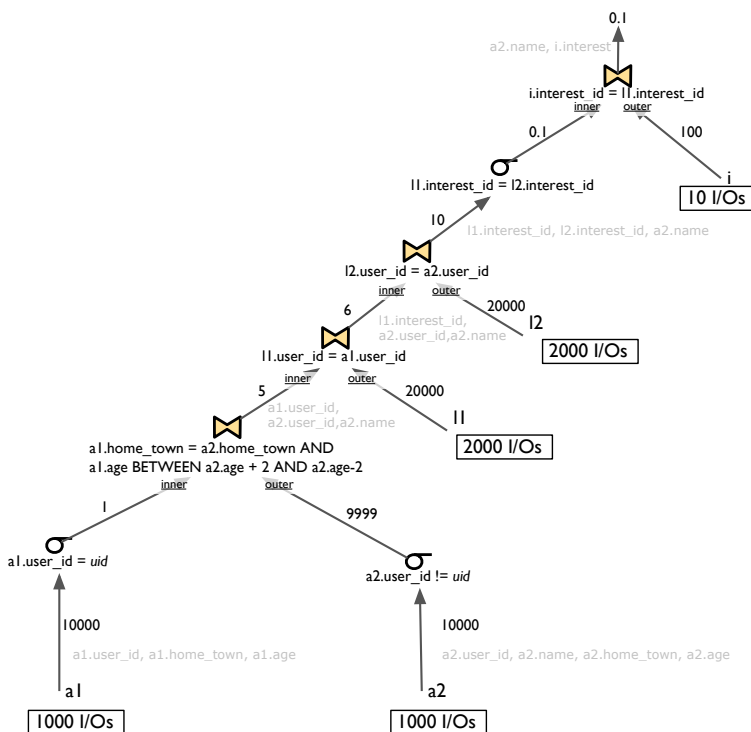


Figure 1: Illustration showing the optimal solution for questions 7, 8, and 9. Edges are labeled with the number of tuples flowing across that edge as well as the fields from each of the tables that are needed (projections are not shown as they do not affect the size of the tuples according to the problem definition.) The only I/Os needed to answer this query are performed at the leaves of the plan.

- A. 3,000
- B. 6,000**
- C. 22,000
- D. 27,000

Name: Solutions

10. [10 points]: Dana decides to add a set of indices to her database. The database system supports clustered and unclustered hash and B+tree indices (with SARGable predicates), and allows each relation to have exactly one clustered index. Disregarding the costs of inserts and deletes to the tables, the best choice of indices from amongst the following options would be:

(Circle ONE.)

- A.** A clustered hash file on `accounts.user_id` and a clustered hash file on `likes.interest_id`
A clustered hash file on `user_id` allows us to avoid one scan of `a1` by directly looking up the `uid` record, saving about 1,000 I/Os – we still have to read almost all of the records from `a2`. The hash file on `interest_id` allows us to avoid the scan of `l2` by taking the output of the join between `l1` and `a1` and finding matching interests in `l2` using the hash file (to evaluate the predicate `l1.interest_id = l2.interest_id`). We can do this before applying the `a2.user_id = l2.user_id` predicate. Thus, these indices save about 3,000 of the 6,000 I/Os over the no-index case.
- B.** A clustered hash file on `accounts.home_town` and a clustered hash file on `likes.interest_id`
Using the hash file on `home_town`, we can look up the 1% of matching users in `a2` after scanning `a1`. So this saves us about 990 I/Os. The hash file on `interest_id` saves us 2,000 I/Os as described above, for a total savings of 2,990 I/Os.
- C.** A clustered B+tree on `accounts.age` and a clustered hash file on `likes.user_id`
We can use the B-tree to find the matching users in `a2` after scanning `a1` – this time, 5% of the users will match, so this will save us about 950 I/Os. The hash file on `likes.user_id`, however, allows us to avoid both scans of `likes`, saving us 4,000 I/Os, so this will save us a total of about 4,950 I/Os.
- D.** A clustered hash file on `accounts.home_town` and a clustered hash file on `likes.user_id`
This is the best combination of the choices listed. The hash file on `home_town` saves 990 I/Os, and the hash file on `likes.user_id` saves 4,000 I/Os for a total savings of 4,990 I/Os.

Concurrency Control

After having selected the indices to use in The Friendinator, Dana starts the system running. She does some profiling of the database and finds that there are many transactions which wait for locks held by other transactions, thus reducing the concurrency (and performance!) of her system. The system uses strict two phase locking, with locks held at the granularity of a tuple, except when an insert is performed in which case the whole table must be locked to prevent other records with conflicting primary keys from being inserted into the table. There are four primary kinds of transactions that run in The Friendinator, as shown below. We have only shown the patterns READ, WRITE, LOCK, UNLOCK, BEGIN, and COMMIT operations performed by each transaction; in general, other computation may be arbitrarily interleaved with these operations. Here “LOCK table_name” indicates a lock on the whole table, and “LOCK table_name.tx” indicates a lock on tuple x of the specified table.

T1: New User Transaction

Adds a new user.

```
BEGIN
X-LOCK accounts
INSERT acct_record
X-LOCK likes
INSERT likes_records
COMMIT
UNLOCK accounts
UNLOCK likes
```

T2: Make Friends Transaction

Adds entries to the `friends` table based on common interests found in the `likes` table.

```
BEGIN
S-LOCK likes.t1
READ likes.t1
...
S-LOCK likes.tn
READ likes.tn
XLOCK friends
INSERT friends_tuple
COMMIT
UNLOCK likes.t1 ... likes.tn
UNLOCK friends
```

T3: Deliver promotion

This transaction finds all users with interests that make them candidates for receiving a particular promotion (e.g., 5% off Red-Sox tickets for users interested in baseball) and updates the status field of those users to indicate that the promotion has been delivered to them.

```
BEGIN
S-LOCK accounts.t1 // scan all accounts
READ accounts.t1
  S-LOCK likes.t1
  READ likes.t1 // determine if this user has an interest
  ... // such that they should receive the promotion
  S-LOCK likes.ti // (by scanning all likes)
  READ likes.ti
...
S-LOCK accounts.ti
READ accounts.ti
  READ likes.t1 // xaction already has S-LOCKS on all likes
  ...
  READ likes.tn
...
X-LOCK accounts.tj //update status field of users j through k
WRITE accounts.tj //who receive the promotion
...
X-LOCK accounts.tk
WRITE accounts.tk
COMMIT
UNLOCK accounts.t1 ... accounts.tn
UNLOCK likes.ti ... likes.tn
```

T4: Find user This transaction corresponds to the SQL query shown above – it finds users who share a common set of interests and are in a particular age range and from the same home town.

```
BEGIN
S-LOCK accounts.t1 // scan all accounts, finding the current user's record
READ accounts.t1
...
S-LOCK accounts.ti
READ accounts.ti
READ accounts.t1 //find other users in the appropriate
... //age range and from the same home town
READ accounts.ti
S-LOCK likes.t1 //find the current user's interests
READ likes.t1
...
S-LOCK likes.ti
READ likes.ti
READ likes.t1 // find users who satisfy age/home
... // town predicate and who share interests
READ likes.ti
S-LOCK interests.t1 //find the relevant interests
READ interests.t1
...
S-LOCK interests.ti
READ interests.ti
COMMIT
UNLOCK accounts.t1 ... accounts.ti
UNLOCK likes.t1 ... likes.ti
UNLOCK interests.t1 ... interests.ti
```

Concurrency Control Questions

11. [5 points]: Which of these transactions have a conflict between their read and write sets, such that in the absence of locking operations they might generate a non-serializable schedule if run concurrently?

(Circle ALL that apply.)

There were two ways to interpret this problem. One is by applying the definition of conflict serializability, which says that two transactions conflict if one reads and the other writes some object, or they both write some object. Using this definition, the following is the correct answer:

- A.** T1 and T1
- B.** T1 and T2
- C.** T1 and T3
- D.** T1 and T4
- E.** T2 and T3
- F.** T2 and T4
- G.** T3 and T4
- H.** T4 and T4

The other possibility is to look at the actual result of the interleaving the queries together, and determine if that result is equivalent to some serial schedule. This produces different results than simply applying the definition of conflict serializability as some transactions whose read and write sets conflict won't actually generate non-serializable schedules due to the semantics of the transactions being run. For example, T3 and T4 "conflict" according to conflict serializability, since T3 updates records of accounts and T4 reads tuples of accounts, but any interleaving of their actions will still result in an equivalent serial schedule because the changes that T3 makes to the records do not affect the results of T4. Under this definition, the only transactions that can result in a non-serial schedule are:

- A.** T1 and T1
- B.** T1 and T2
- C.** T1 and T3
- D.** T1 and T4
- E.** T2 and T3
- F.** T2 and T4
- G.** T3 and T4
- H.** T4 and T4

12. [7 points]: Dana suspects that deadlock may play a role in the performance problems she observes. Which of these transactions could result in a deadlock if run concurrently?

(Circle ALL that apply.)

All of transactions order the locks which they acquire, such that deadlocks should never occur. Solutions in which workloads with T3 and T4 were circled were also accepted, since it's not clear that T3 acquires X-LOCKS on accounts in order. In this case, if a T4 transaction S-LOCKS account i, and a T3 transaction X-LOCKS account j where $j > i$, and T4 then tries to access account j while T3 tries to access account i, deadlock will occur.

- A. T1 and T2
- B. T1 and T3
- C. T1 and T4
- D. T2 and T3
- E. T2 and T4
- F. T3 and T4
- G. T1, T2, and T3
- H. T1, T2, and T4
- I. T2, T3, and T4
- J. T1, T2, T3, and T4

Dana's friend R.D. Onlee notices that most of the transactions that run in her system are of type T4, and theorizes that she may get better performance if she uses optimistic concurrency control. Fortunately, the database system The Friendinator is built with can be configured to run in an optimistic mode, which she determines is similar to the optimistic concurrency control method using the Parallel Validation scheme proposed by Kung and Robinson. When the above transactions are run with optimistic concurrency control turned on, none of the X-LOCK, S-LOCK, or UNLOCK statements are included. The code for the Parallel Validation Scheme from the Kung and Robinson paper is reproduced at the end of this exam.

13. [12 points]: 85% of the transactions run on The Friendinator are type T4, and 5% are type T1, 5% type T2, and 5% type T3. All transactions take about the same amount of time to run and the users involved in transactions T2, T3, and T4 are uniformly and randomly selected. At any one time, about 40 transactions are running simultaneously. You may assume that all 40 transactions are issued at the same instant in time (i.e., that the $startT_n := tnc$ statement in the `tbegin` method of the optimistic concurrency protocol happens at the same instant on all transactions.) Estimate the fraction of transactions of type T4 that must be restarted at least once by the validation phase of the optimistic concurrency control protocol in The Friendinator.

(Circle the best answer.)

A. < 10 %

In the parallel validation scheme, the optimistic concurrency protocol will abort a transaction of type T4 if its read set intersects the write set of a transaction that has its transaction number assigned earlier than it did, or if its read or write set intersects the write set of any transaction doing parallel validation at the same time as it is (note that this second condition does not matter since T4 has no write set.) T4 can conflict with transactions T1 and T3, since its read set intersects their write set. At any given time, 10% of the running transactions are type T1 or T3. If we assume the 40 concurrent queries complete in a randomly selected order, the expected position of the first transaction of type T1 or T3 is $1/P(T1 \text{ or } T3) = 10$. Thus, about 9 transactions of type T4 or T2 will complete before the first transaction of type T1 or T3 completes, and of those, 34 out of 36, or 8.5 out of 9 will be T1 (the other 2 will be type T2). After the first T1 or T3 completes, the remaining transactions of type T4 must abort, since they may have read data that the transaction of type T1 or T3 dirtied. This means that of the $.85 \times 40 = 34$ transactions of type T4 running, we expect that 8.5 will complete and the remaining 34 will abort. $8.5/34 = .75$ – thus, about 75% of the transactions of type T4 will abort at least once.

B. 20 %

C. 30 %

D. 50 %

E. > 75 %

As we computed above, this is the correct answer.

Unfortunately, when Dana tries running her system with optimistic concurrency control on, she finds that the performance of her system decreases dramatically versus when it is run with two-phase locking.

14. [10 points]: The following are likely explanations for this substantial performance decrease:
(Circle ALL that apply.)

- A.** The ratio of read-only (e.g., type T4) to read/write transactions isn't high enough, and thus the benefits of optimistic concurrency control are lost.

As we discussed in class, a mostly read-only workload doesn't mean that optimistic concurrency control will perform better than a read/write workload, since locking based protocols are low-overhead in read-only scenarios as well. Thus, although it's true that increasing the ratio of read-only transactions would improve the performance of the optimistic concurrency protocol, it would also improve the performance of the locking protocol.

- B.** The Friendinator is running on a single processor machine, and the additional CPU overhead of restarting transactions in optimistic concurrency control impairs its performance.

This would explain the performance loss in The Friendinator – as the paper on Concurrency Control Performance Modeling points out, optimistic concurrency control doesn't do well when there are limited resources, since restarting transactions uses more CPU and disk than simple locking.

- C.** Locks can be acquired and released much faster in a locking protocol than conflict detection can be done in the validation phase of optimistic concurrency control.

This is not true – neither lock acquisition nor validation is particularly expensive – most differences in the protocols are due either to blocking due to locking or wasted resources due to restart in optimistic concurrency control.

- D.** The locking protocol is probably using table-level locks, while the optimistic method is validating accesses on a per-tuple level. Thus, locking is much lower overhead.

Table-level locking is unlikely to improve the performance of a locking-based system, since many more transactions will have to wait for locks. So this is untrue.

- E.** Optimistic concurrency control requires extensive use of transaction UNDO to abort running transactions, and using the recovery manager so frequently makes the overhead of optimistic concurrency control much higher.

This is true, for the same reason that B is true.

Optimistic Concurrency Control Code – Parallel Validation

```
tcreate = (  
  n := create  
  createSet := createSet ∪ {n}  
  return n)  
  
twrite(n,i,v) = (  
  if n ∈ createSet then  
    write(n,i,v)  
  else if n ∈ writeSet then  
    write(copies[n],i,v)  
  else (  
    m := copy(n)  
    copies[n] := m  
    writeSet := writeSet ∪ {n}  
    write(copies[n], i, v)))  
  
tread(n,i) = (  
  readSet := readSet ∪ {n}  
  if n ∈ writeSet then  
    return read(copies[n], i)  
  else  
    return read(n,i)  
  
tdelete(n) = (  
  deleteSet := deleteSet ∪ {n})  
  
tbegin = (  
  createSet := empty  
  readSet := empty  
  writeSet := empty  
  deleteSet := empty  
  startTn := tnc)  
  
tend = (  
  <finishTn := tnc  
  finishActive := (make a copy of active)  
  active := active ∪ {id of this transaction}>  
  valid := true  
  for t from startTn + 1 to finishTn do  
    if (writeSet of transaction t intersects readSet) then  
      valid := false  
  for i ∈ finishActive do  
    if (writeSet of transaction i intersects readSet or writeSet) then  
      valid := false  
  if (valid) then (  
    (write phase)  
    <tnc := tnc + 1  
    tn := tnc  
    active := active - {id of this transaction}>  
    (cleanup)  
  ) else (  
    <active := active - {id of this transaction}>  
    (backup)))
```