Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.830 Database Systems: Fall 2005

Quiz I Solutions

There are 15 questions and 12 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.* 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.

Do not write in the boxes below

1-4 (xx/24)	5-7 (xx/18)	8-9 (xx/9)	10-12 (xx/21)	13-15 (xx/28)	Total (xx/100)

Name: Solutions

I Short Answer Reading Questions

The paper by Chou and DeWitt on buffer management strategies argues that a simple LRU buffer management policy is not the best choice in a database system. Suppose you have a buffer pool of 100 pages, and that each page can hold 10 tuples worth of data. Further, suppose the following operations are concurrently executing in your system:

A. A sequential scan of a relation, R, consisting of 10,000 tuples.

B. A nested loops join of a relation, S consisting of 1,000 tuples with a relation, T, consisting of 500 tuples, with S as the outer relation.

C. An index nested loops join of a relation U, consisting of 1,000 tuples, with a relation V, consisting of $10,000$ tuples, with V as the inner. Suppose there is a primary B+-Tree index on V, and that the B+-Tree has branching factor of 10 and fill factor of 0.5. Each lookup in V touches exactly one leaf (data) page.

1. [6 points]: Estimate (to the nearest 100) the number of pages that an LRU buffer management policy would read from disk when processing these operations concurrently, assuming that each operation takes about the same amount of time to complete and begins at about the same time:

(Show your work in the space below.)

Since all of the operations take the same time to run, during each pass over the inner relation of operation B, operation A will scan 1/1000th of its tuples, or 1 page worth of data. Similarly, for each outer tuple processed in B, C will process one outer tuple as well; this will involve a single lookup in the B-Tree, which is 5 levels (so 5 pages will be accessed.) Depending on how you chose to interpret the "fill factor or .5" requirement, you might also have concluded that the B-Tree would be 6 levels.

For simplicity, let's assume that these operations run in an interleaved fashion such that one page is processed by A, then one outer tuple is processed by B, then one outer tuple is processed by C.

Hence, the cost of each of the three operations is as follows:

A. *1,000 pages will be read*

B. *100 pages of the outer will be read. Each pass of the inner will access 50 pages. The first pass will require us to reread all of the inner. On successive passes of the inner, A will have evicted one other page, and C will have evicted 5 other pages. But since B reaccesses each of its 50 pages on for every outer tuple processed by A and C, its pages won't ever be evicted. Hence, only 150 pages are read.*

C. *100 pages of the outer will be read. Each will incur 5 lookups in the B-tree for the inner. Clearly, the top page of the B-Tree will remain in the buffer pool. Each 2nd level page in the B-Tree will be re-accessed, on average, one out of every ten lookups. After ten lookups, A will have accessed ten pages, and B will have accessed one new page of inner tuples. Hence, with high probability, the top 2 levels of the B-Tree will remain in cache. Lower levels of the B-tree are unlikely to be cached since they are re-accessed only 1 out of 100 outer tuples at most. The total cost is 100 + 1 (top level) + 10 (2nd level) + 3 * 1000 = 3111 pages.*

Hence, the total number of pages read by the LRU approach is: $3111 + 150 + 1000 = 4261$ *pages.*

Name:

2. [6 points]: Estimate (to the nearest 100) the number of pages that the DBMIN buffer management policy (proposed in the Chou and DeWitt paper) would read from disk when processing these operations concurrently, assuming that each operation takes about the same amount of time to complete and begins at about the same time, and that buffer pool pages are optimally allocated to each operator.

(Show your work in the space below.)

DBMIN would allocate 1 page to A), 50 pages to B) and the remaining 49 pages to C).

- **A.** *will access 1,000 pages.*
- **B.** *will access 150 pages, as above.*
- **C.** *will cache the top 2 levels of the tree in 11 pages. The remaining 38 pages will be managed with an MRU policy, which will result in 38% of the 100 third level pages being cached after the first lookup. All of the level 4 and 5 pages will need to be re-read on each iteration.*

Hence, the total cost is:

$1,000 + 150 + 1 + 10 + 100 + 2.62 * 1000 = 2731$ *pages.*

Ben Bitdiddle is designing a database to store his grades in his classes at MIT. He chooses the following schema:

```
grades
 {      studentname : string //e.g., Ben
        department : int //e.g., 6
        number : int  //e.g., 830
        grade : char //e.g., A, B, C, D, F
        professor : string //e.g., Madden
        year : int //e.g., 2005
        term : string //e.g., Fall
 } primary key (studentname, department, number, year, term)
```

3. [6 points]: List three specific examples of problems (anomalies) that might arise when inserting into, deleting from, or updating this database.

- **A.** *Multiple students might take the same class on the same year/term, resulting in duplicated data and the possibility for insertion/update anomalies.*
- **B.** *If all students drop a class, no information about that class will be in the database.*
- **C.** *If the same class is offered on different years / terms, information about that class (e.g., its department and number) will be recorded multiple times, allowing insertion/update anomalies.*

Name:

4. [6 points]: Propose a decomposition of Ben's schema into two or more tables that avoids the problems you listed above.

(Show your schema below.)

```
grades
{       studentname : string
        catalog_id : int references catalog.catalog_id
        grade : char
} primary key (studentname, catalog_id)

course
{      course_id : int primary key
       department : int
       number : int
}


catalog
{     catalog_id : int primary key
      course_id : int references course.course_id
      prof_id : int references professors.prof_id
      year : int
      term : string
}

professors
{     prof_id : int primary key
      professor : string
}
```

Name:

Views

5. [6 points]: Views are one of the most useful constructs in database systems. List two important uses for them:

(Limit your answer to one sentence per use.)

A. *Views enable logical independence by allowing schema changes to be while still providing backwards compatibility with existing applications written against the old schema.*

B. *Views provide a security mechanism that allows users to only access the portions of the database they are authorized to read.*

6. [6 points]: Under what circumstances might you choose to materialize (i.e., precompute and store the contents) a view?

(Limit your answer to one or two sentences.)

Materialized views are a good idea when the costs of applying a view rewrite are high. Hence, if the view is complex and being accessed frequently, it is a good candidate for materialization.

7. [6 points]: Should all views be materialized? If not, what is the downside (relative to regular, unmaterialized views) of creating a materialized view?

(Limit your answer to one or two sentences.)

No. It is a bad idea to materialize views on relations that are updated frequently, because materialized views must be recomputed after each update.

Recovery

Ben Bitdiddle, after reading the ARIES paper, claims that it would be more efficient to use logical REDO processing rather than physical REDO processing as proposed in the ARIES paper, since logical records are more compact and will require fewer disk I/Os than physical log records. Dana Bass explains to Ben that he is wrong on two accounts: first, ARIES will not work with logical REDO logging, and second, there are some situations in which physical logging might be more efficient than logical logging.

8. [6 points]: Describe, in one or two sentences, why ARIES requires physical REDO.

(Write your answer in the space below.)

ARIES requires physical REDO because the system may be in an inconsistent state after a crash. Because pages can be arbitrarily corrupted, logical REDO may not be able to restore the state of a page to a correct state, whereas physical log records that capture they exact bytes in a page would be able to do this. Physical REDO also guarantees idempotency, which may be important during repeated invocations of recovery.

9. [3 points]: Describe, in one or two sentences, a situation in which physical logging would outperform logical logging.

(Write your answer in the space below.)

Because logical logging requires an operation to be applied, doing this may be more expensive copying the new bytes into place. Logical logging may also require updates to be done in several places (e.g., rebalance a B-Tree, etc.); physical logging would log each of these physical updates separately and would not have to do any computation to determine how to apply the changes.

Name:

II Sir Votes-a-Lot

Fed up with her perception of disenfranchisement of voters during the recent elections, Dana Bass is building a new Linux-based voting machine called "Sir Votes-a-Lot" that is designed to be more secure and accurate than previous voting machines. She plans on using an open source database to store the data about votes, polling places, candidates, and other election data. Your job is to help her with performance issues of her database system. Because we have not studied distributed databases yet, you should simply assume that votes from different polling places are transmitted over the Internet to a single Sir Votes-a-Lot database that collects all of the voting information.

Dana decides that the database used in Sir Votes-a-lot will consist of six tables: a `voter` table that has information about each voter (including a reference to the place where he or she votes), a `place` table that has information about each polling place, a `contest` table that has information about each position that is up for grabs in the election, a `candidates` table that contains information about each candidate, an `eligibility` table that lists the locations (as listed in the `place` table) that can vote in each contest, and a `votes` table that records the votes of each voter in each contest.

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

```
voters
 {      id : int primary key,
        name : string
        sex : char,
        age : int,
        place: foreign key references places.place_id
 }
```

This table keeps track of the voters in the system. Place is a reference to the location in which the voter votes.

```
places
 {       place_id : int primary key,
         district : string
         city : string
         state : string
 }
```

This tables lists places where voting happens, as defined by a particular district inside of a city. For simplicity, each contest is associated with one or more places via the `eligibility` table, and each voter is associated with exactly one place.

```
contest
 {     contest_id : int primary key
       title : string

 }
```

This table contains information about the different positions being contested during this election, where title is the position being elected (e.g., President).

Name:

```
 candidates
 {      candidate_id : int primary key
        contest : int foreign key references contest.contest_id
        name : string
        age : int
        party : string
}
```

This table contains information about candidates and the positions they are running for.

```
eligibility
 {      place : int foreign key references places.place_id
        contest : int foreign key references contest.contest_id
 }  primary key place, contest
```

This table lists the places eligible to vote in each contest.

```
votes
 {      voter : int foreign key references voters.id
        contest : int foreign key references contest.contest_id
        candidate : int foreign key references candidates.candidate_id
 }  primary key voter, contest, candidate
```

This table lists the candidate that each voter voted for in each contest.

Query Optimization

Because she is particularly interested in younger candidates, Dana wants to compute the count of votes obtained by candidates under the age of 30 who are being voted on by voters in Massachusetts. She writes the following query:

```
SELECT candidate.name, contest.title, COUNT(*)
 FROM candidate, contest, votes
 WHERE votes.contest = contest.contest_id
 AND votes.candidate = candidate.candidate_id
 AND candidate.age < 30
 AND contest.contest_id IN
        (SELECT DISTINCT contest_id //contests in MA
        FROM eligibility, places
        WHERE eligibility.place = places.place_id
        AND places.state = ''MA''
        )
 GROUP BY candidate.name, contest.title
```

Dana asks you to help estimate the cost and improve the performance of this query. At first, she has no indices. Assume there are 5,000 candidates, 100,000 voters, 1,000 contests, 1,000 places, that each contest is eligible to be voted on at 10 places, and that each voter votes in all contests he or she is eligible for.

Name:

10. [6 points]: Suppose that each of the 50 US states has the same number of voters, and that the age of voters is evenly distributed between 18 and 118. Further suppose that there is no correlation between candidate age and the number of votes he or she receives. Estimate the selectivity of the following predicates, where selectivity is between 0 and 1, and the selectivity of a join is the fraction of the cross product of the two input relations that is output:

(Estimate the selectivity for each of these expressions:)

A. The selection (IN expression) on contest.contest_id: $\frac{1}{50}$

B. The selection on candidate.age: $\frac{12}{100}$

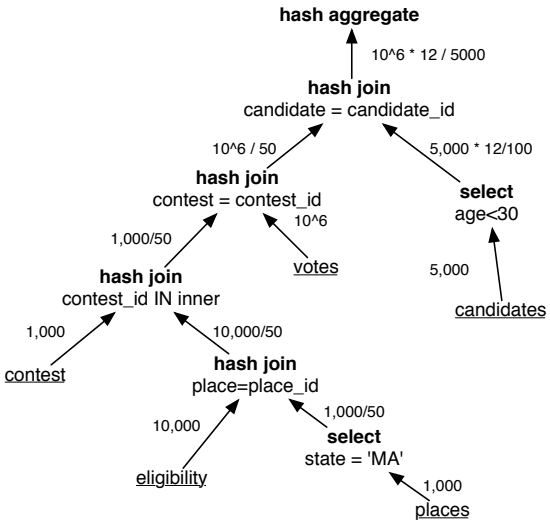
C. The join on votes.contest and contest.contest_id: $\frac{1{,}000{,}000}{(1{,}000{,}000)(1{,}000)}$

D. The join on votes.candidate and candidate.candidate_id: $\frac{1{,}000{,}000}{(1{,}000{,}000)(5{,}000)}$

E. The join on eligibility.place and places.place_id: $\frac{10{,}000}{(1{,}000)(10{,}000)}$

11. [10 points]: In the space below, sketch a query plan for this query that would result in the minimum amount of work given no indices and the selectivity estimates you gave above. Be sure to indicate what join algorithm you would use for each join as well as which relation is the inner/outer in any nested loops joins. Assume you have sufficient memory to fit all relations and any intermediate data structures in memory.

(Show your plan in the space below.)



At the bottom of the plan, we compute the subquery and join its results with the contest table (to defer joining with the large votes table as much as possible.) We choose to order the remaining joins such that as much of the votes table is filtered out as early as possible. Hence, because the join with contest_id's in MA is most restrictive, we apply it first. The second most selective join is with candidate, so we apply it next. All selections are pushed down as far as possible.

Name:

12. [5 points]: Dana knows she should add some indices to her database. Recommend a set of indices that will improve the performance of this query the most, assuming the database supports clustered and unclustered Hash and B+-tree indices and that at most one clustered index can be created per relation. Justify your choice.

(Show you answer in the space below.)

Assuming all relations and indices can fit into memory, she should create:

- *A clustered hash index on places.state to speed up the selection on places.*
- *A clustered hash index on eligibility.place_id to use as the inner relation of an index-nested loops join with places, to avoid having to read all of eligibility (since only 1/50 needs to be read.)*
- *A clustered hash index on votes.contest_id, to use as the inner relation of an index-nested loops join with, to avoid having to read all of votes.*
- *A clustered B-tree index on candidates.age, to speed up the selection on age.*
- *(Possibly) a clustered hash index on contest.contest_id to be used as the inner relation of an index-nested loops join on with the inner query. This will avoid reading information about all of the contests when only a small fraction of them are actually accessed.*

Note that using index-nested loops joins will result in random I/O (instead of sequential I/O as in hash join), but that the reduction in the number of records that has to be read for most of these joins will make this tradeoff worthwhile.

Concurrency Control

After Dana has deployed Sir Votes-a-lot, she notices that the performance of her system is poor. She does some profiling of the system system and discovers that the bottleneck is in the locking system. The database system workload consists mainly of two types of transactions (here, `W, X, Y` and `Z` refer to constants that are written as a part of the query by the issuing program):

T1:

```
BEGIN TRANSACTION
voter_id = SELECT voters.id FROM voters WHERE voters.name = X
INSERT INTO votes VALUES (voter_id, W_1, Z_1)
...
INSERT INTO votes VALUES (voter_id, W_n, Z_n)
COMMIT
```

T2:

```
BEGIN TRANSACTION
total = SELECT count(*) FROM votes WHERE contest = X
SELECT candidate, count(*)/total FROM votes
       WHERE contest = X GROUP BY candidate
COMMIT
```

Name:

At any given time, there are about 100 times as many T1 transactions running as T2 transactions. Suppose that each T1 transaction runs 10 inserts on average and that those inserts are evenly distributed over candidates, contests, and voters. Assume that the contests referenced by the set of T2 transactions are evenly distributed over all contests. Finally, assume that (for the purposes of these queries), Dana has created a clustered hash index on voters.name and on votes.contest (so that the WHERE clause in the SELECT statements can be satisfied without doing a table scan.)

13. [16 points]: Dana wants to understand how different degrees of locking and degrees of consistency will affect the performance of her database. Suppose a page in the database holds 100 records. Using the table cardinalities given above (5,000 candidates, 100,000 voters, 1,000 contests, 1,000 places, that each contest is eligible to be voted on at 10 places, and that each voter votes in all contests he or she is eligible for), estimate the number locking requests per transaction of type T1 and T2 for the different degrees of consistency and granularity of locks listed below. If a transaction re-requests a lock it already holds, that still counts as a locking request.

(Write your answer in the space below.)

A. Degree 3 consistency, record-level locking:
No. of lock manager requests per T1: *1(S), 11(X)*
No. of lock manager requests per T2: *1,000,000 votes, 1,000 contents implies 1,000 locks/contest*

B. Degree 3 consistency, page-level locking:
No. of lock manager requests per T1: *1(S), 1(X), assuming all inserted records are on same page*
No. of lock manager requests per T2: *Because votes is clustered on contest, assume that all 1,000 records are co-located. Hence, only 10 locks are needed.*

C. Degree 1 consistency, record-level locking:
No. of lock manager requests per T1: *10(X)*
No. of lock manager requests per T2: *0*

D. Degree 1 consistency, page-level locking:
No. of lock manager requests per T1: *1(X)*
No. of lock manager requests per T2: *0*

Name:

14. [9 points]: Dana concludes that switching her database to degree 1 consistency with record level locking will help reduce lock contention (e.g., the number of times a transaction has to wait to acquire a lock), which she believes is the major performance bottleneck with Sir Votes-a-Lot. With the mix of T1 and T2 transactions given above , will degree 1 consistency have significantly less lock contention than degree 3 consistency with record level locks? Why or why not? Provide an analytical justification for your answer.

(Write your answer in the space below.)

Degree 1 consistency will have significantly less lock contention. If at any given time there are 100 T1 transactions and 1 T2 transaction running, the T1 transactions will have 1100 locks, and the T2 transactions will have 1000 locks when running at level 3 consistency. At level 1 consistency, only 1000 locks will be needed by the T1 transactions (T2 transactions need no locks). Hence, there is a significant reduction in lock pressure.

15. [3 points]: What are the other implications of using degree 1 consistency?

(Write your answer in the space below.)

- *T2 transactions may see a different value for* `total` *than the sum of all the votes in contest* `X`*, since it reads dirty data.*
- *If T1 transactions abort with any frequency, T2 transactions may see larger errors as they read uncommitted data.*

End of Quiz I

Name: