Name:_____

CSE 544, Fall 2006, Take-Home Examination
Due: 13 November 2006, in class

Rules:

- You can use any documentation that you want to complete this exam.
- You are allowed to search for information on the Web.
- Please write clearly.
- **You are not allowed to talk to anyone about the exam.**

Question	Max	Grade
1	10	
2	15	
3	25	
4	10	
5	20	
6	12	
7	8	
Total	100	

Name:_____

1. (**10** points) **Data Independence**

   (a) (**2** points) What is physical data independence?

   Solution:
   Physical data independence is a property of a DBMS that ensures decoupling between the physical layout of data and applications which access it. In other words, with physical independence, applications are insulated from changes in physical storage details: changes in how the data is stored do not cause application changes.

   (b) (**1** points) What properties of the relational model facilitate physical data independence?

   Solution:
   Declarative query language or set-at-a-time query language.

   (c) (**2** points) What is logical data independence?

   Solution:
   Logical data independence is a property of a DBMS that ensures decoupling between the logical structure of data and applications that operate on it. With this property, changes in the logical data layout like tables, rows, and columns do not require application to be changed.

Name:_____

(d) (**1** points) How can one provide a high level of logical data independence with the relational model?

Solution:
By defining views.

(e) (**4** points) Name four concepts introduced by E. F. Codd in his paper "A Relational Model of Data For Large Shared Data Banks" that are part of the relational model as we know it today.

Solution:
Relational algebra, Join, Projection, Constraints (primary key, foreign key), Set-at-a-time query language, ...

Name:_____

2. (**15** points) **Schema normalization**

Your friend Bob is designing a database for a scientist. The database will serve to track tagged animals over time. Sensors placed at fixed locations will automatically record animal sightings. Bob is considering using the following schema:

```
Animals (
            tag_id : integer,      Unique ID associated with the animal's tag
            type: string,          The type of animal: cow, crow, turtle, etc.
            description: string,   Describes any special characteristics of this type of animal.
            markings: string,      Describes any special markings of this animal.
            sighting_date: date,   The date when the animal was detected by the sensor.
            sensor_id: integer,    Sensor ID.
            latitude: integer,     Location of the sensor.
            longitude: integer     Location of the sensor.
)
```

where `(tag_id, sighting_date, sensor_id)` is the primary key of `Animals`.

(a) (**5** points) List and describe three specific examples of problems (anomalies) that might arise when inserting into, deleting from, or updating this database.

Solution:

- Insertion anomaly: an insertion anomaly occurs when it is not possible to store certain information unless some other, unrelated, information is stored as well. In the example above, we cannot add a new animal without at least one sighting of that animal or we cannot add a new sensor without at least on sighting by that sensor because the primary key includes a `tag_id`, a `sighting_date`, and a `sensor_id`.
- Deletion anomaly: a deletion anomaly occurs when it is not possible to delete certain information without losing some other, unrelated, information as well. In the example above, we cannot remove (reset) all sightings of an animal without removing the animal from the table.
- Update anomaly: an update anomaly occurs if updating one copy of repeated data creates an inconsistency unless all copies are similarly updated. In the example above, if the markings of an animal need to be changed, every tuple with the same tag_id must be updated as well.

(b) (**5** points) What is the difference between the following two normal forms: 3NF and BCNF? [Please continue on next page]

Solution:

A relation R is in BCNF if, for every FD $X \rightarrow A$ that holds over R, one of the following is true:

- $A \in X$ (trivial dependency)
- X is a superkey

A relation R is in 3NF if, for every FD $X \rightarrow A$ that holds over R, one of the following is true:

- $A \in X$ (trivial dependency)
- X is a superkey
- A is part of some key for R.

BCNF is thus more strict than 3NF. A relation in BCNF is also in 3NF, but not the other way around. Indeed, a relation R is in 3NF, even when a functional dependency of the following form holds over that relation: $X \rightarrow A$, where X is NOT a superkey of R, but A is part of some key

for R. Such a functional dependency cannot hold over a relation in BCNF. In BCNF, X must be a superkey for all FDs that hold over the relation (or a functional dependency must be a trivial functional dependency where $A \in X$).

These two normal forms have the following property:

	3NF	BCNF
Preserving dependency	Yes	No
May have redundancy	Yes	No

BCNF guarantees that no redundancy can be detected using FD information alone. It is thus the more desirable form of the two. However, it is not always possible to decompose a relation into BCNF without losing information about some dependencies. 3NF makes an exception for some FDs (see above) to ensure that any relation can be decomposed into 3NF using a lossless-join, dependency-preserving decomposition.

(c) (**5** points) Propose a decomposition of Bob's schema into multiple tables that avoids the problems you listed above (show your schema below).

Solution:

```
AnimalTypes (
               type:  string,
               description:  string,
)
Animals (
           tag_id:  integer,
           type:  string,
           markings:  string
)
Sensors (
           sensor_id:  integer,
           latitude:  integer,
           longitude:  integer
)
Sightings (
             sighting_date:  date,
             tag_id:  integer,
             sensor_id:  integer,
)
```

Name:_____

3. (**25** points) **Query optimization**

Given the following SQL query:

```
Student (sid, name, age, address)
Book(bid, title, author)
Checkout(sid, bid, date)

SELECT S.name
FROM   Student S, Book B, Checkout C
WHERE  S.sid = C.sid
AND    B.bid = C.bid
AND    B.author = 'Olden Fames'
AND    S.age > 12
AND    S.age < 20
```

And assuming:

- There are $10,000$ `Student` records stored on $1,000$ pages.
- There are $50,000$ `Book` records stored on $5,000$ pages.
- There are $300,000$ `Checkout` records stored on $15,000$ pages.
- There are 500 different authors.
- Student ages range from 7 to 24.

(a) (**5** points) Show a physical query plan for this query, assuming there are no indexes and data is not sorted on any attribute.

Solution:

Note: many solutions are possible.

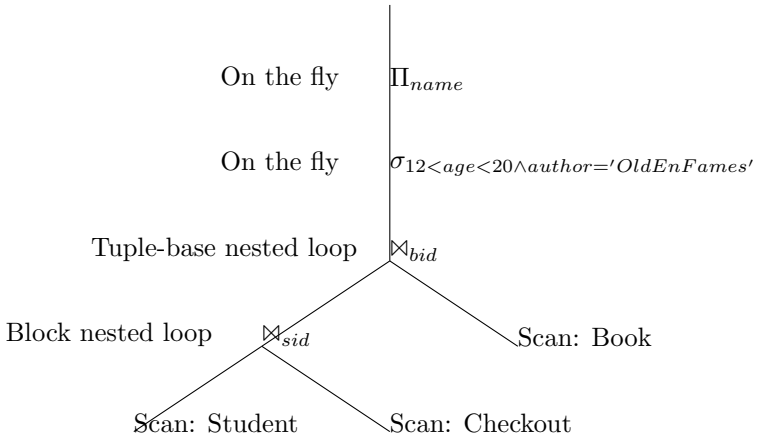

Figure 1: One possible query plan (all joins are nested-loop joins)

Name:_____

(b) (**5** points) Compute the cost of this query plan and the cardinality of the result.

Solution:

	Cost	Cardinality	Remarks
$S \bowtie C$	B(S) + B(S) * B(C) = 1000 + 1000 * 15000 = 15001000	300000	(1)
$(S \bowtie C) \bowtie B$	T($S \bowtie C$) * B(S) = T(C) * B(S) = 300000 * 5000 = 1500000000	300000	(2)
σ and Π	On the fly	300000 * σ_{author} * σ_{age} = 300000 * $\frac{1}{500}$ * $\frac{7}{18}$ $\approx$ 234	(3)
Total	1515001000	234	

(1) We are doing page at a time nested loop join. Also, the output is pipelined to next join.
(2) The output relation is pipelined from below. Thus, we don't need the scanning term for outer relation.
(3) We assume uniform value distributions for age and author. We assume independence among participating columns.

(c) (**5** points) Suggest two indexes and an alternate query plan for this query.

Solution:

Note: many solutions are possible.

We assume there are only two indexes for this query: an unclustered B+-tree index on `Book.author` and a clustered B+tree index on `Checkout.bid`:

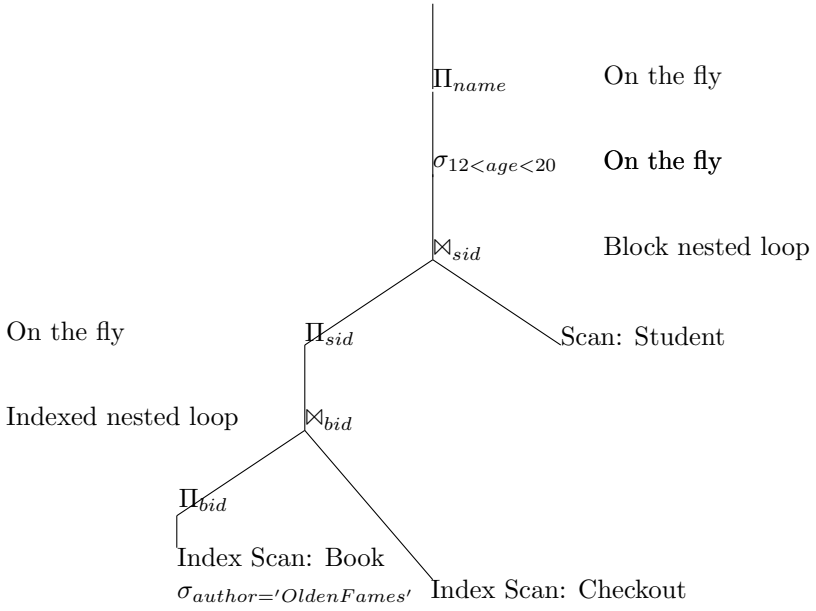

Figure 2: One possible query plan that uses the two indexes

Name:_____

(d) (**5** points) Compute the cost of your new plan.
Solution:

- $N(B)$ = # of tuples per page for `Book` = T(B)/B(B) = 10
- $N(C)$ = # of tuples per page for `Checkout` = T(C)/B(C) = 20

	Cost	Cardinality	Remarks
Index Scan on Book with σ_{author}	T(B) * 1/V(B) $= 50000 * \frac{1}{500}$ $= 100$	100	(1)
$\Pi_{sid}(B \bowtie C)$	$100 * \lceil (T(C)/V(bid))/N(C) \rceil$ $= 100 * \lceil (300000/50000)/10 \rceil$ $= 100$	600	(2)
$\bowtie_{sid}$	B(S) = 1000		(3)
Total	1200		

(1) We assume all intermediate index pages are in memory. Note: because `bid` is the search-key for the index, the `bid` values are in the leaf pages of the index: they are thus in memory as per the first assumption. Because we project on `bid` right after the selection, we only need these values, so we do not really need to perform any disk I/Os.
(2) One index lookup per outer tuple. Assuming uniform distribution, there will be 6 checkouts per book. Assuming all intermediate index pages are in memory, the 6 records can be fetched with only one or two disk accesses since we have a clustered index on `Checkount.bid`. The above computation is optimistic but it only incurs 100 more I/Os in the worst case.
(3) Again, the output of the previous operation is projected on *sid*. Because there are only 600 tuples, it is reasonable to assume all results can hold in memory. Since the outer relation is already in-memory, we only need to scan the inner relation `Student` one time.

The above plan thus dramatically reduces the number of I/Os compared with the naive plan in (*a*). However, having a clustered index on `Checkout.bid` is unlikely in real situations because it would result in high insert costs.

(e) (**5** points) Explain the steps that the System R query optimizer would take to optimize this query.
Solution:
A query optimizer explores the space of possible query plans to find the most promising one. The System R query optimizer performs the search as follows:

- Only considering left-deep query plans.
  Instead of enumerating all possible plans and evaluating their costs, the optimizer keeps the efficient pipelined execution model in mind. Thus, it only looks for left-deep query plans and enumerates different join orders. It considers cartesian products as late as possible to reduce I/O costs. It considers only nested-loop and sort-merge joins.
- in bottom-up fashion.
  The optimizer starts by finding the best plan for one relation. It then expands the plan by adding one relation at a time as an inner relation. For each level, it keeps track of the cheapest plan per interesting output order, which will be explained shortly, as well as the cheapest plan overall. When computing the cost of a plan, System R considers both I/O cost and CPU cost.
- considering interesting orders.
  If the query has an ORDER BY or a GROUP BY clause, having results ordered by the

columns that appear in those clauses can reduce the cost of the query plan because it can save extra I/Os needed by sort or aggregation. Similarly, attributes that appear in join conditions are considered interesting orders because they reduce the cost of sort-merge joins. When the System R optimizer evaluates a plan, at each stage, it keeps track of the cheapest plan per interesting order in addition to the cheapest plan overall.

Name:_____

4. (**10** points) **Transactions: concurrency control**

Consider a database with objects X and Y and assume that there are two transactions T_1 and T_2. T_1 first reads X and Y and then writes X and Y. T_2 reads and writes X then reads and writes Y.

(a) (**3** points) Give an example schedule that is not serializable. Explain why your schedule is not serializable.

Solution:
$R_1[X] \rightarrow R_1[Y] \rightarrow R_2[X] \rightarrow W_2[X] \rightarrow R_2[Y] \rightarrow W_1[X] \rightarrow W_1[Y] \rightarrow W_2[Y] \rightarrow C_1 \rightarrow C_2$

A schedule is serializable if it contains the same transactions and operations as a serial schedule *and* the order of all conflicting operations (read/writes to the same objects by different transactions) is also the same. In the above schedule, T_1 reads X before T_2 writes X. However, T_1 writes X after T_2 reads and writes it. The schedule is thus clearly not serializable. Additionally, according to the above schedule, the final content of object X is written by T_1 and the final content of object Y is written by T_2. Such a result is not possible in any serial execution, where transactions execute one after the other in sequence.

(b) (**3** points) Show that strict 2PL disallows this schedule.

Solution:

Strict 2PL has two two rules:

i. If a transaction T wants to read (respectively, modify) an object, it first requests a shared (respectively, exclusive) lock on the object.

ii. All locks held by a transaction are released when the transaction is completed.

With strict 2PL, the above schedule is not possible. Indeed, T_1 first acquires shared locks on X and Y. When T_2 runs, it also acquires a shared lock on X. When it tries to acquire an exclusive lock before writing X, however, it blocks, waiting for T_1 to release its lock, which will happen only when T_1 commits. The above schedule is thus impossible with strict 2PL. In fact, the schedule leads to a deadlock: when T_1 tries to write X, it also blocks waiting for T_2 to release its shared lock. Now, both transactions are waiting for each other. We have a deadlock. When the deadlock is detected, the DBMS will abort one of the transactions, allowing the other one to commit and release its locks.

Name:_____

(c) (**4** points) What are the differences between the four levels of isolation?
Solution:

- **Read uncommitted:** Transactions do not need to acquire any locks before reading data. Transactions may thus read data written by other transactions that have not yet committed. The value read may thus later be changed further or rolled-back. This problem is called the *dirty read* problem. This level of isolation also suffers from all the problems of the more restrictive isolation levels below.
- **Read committed:** Transactions must acquire shared locks before reading data. They may release these locks as soon as they read the data (short duration read locks). This level of isolation guarantees that the transaction never reads uncommitted data by other transactions. However, it doesn't ensure the data will not change until the end of the transaction. If a transaction reads the same data item twice, it can see two different values. This problem is called the *non-repeatable read* problem. This level of isolation also suffers from all the problems of the more restrictive isolation levels below.
- **Repeatable read:** Transactions must acquire long duration read locks on the individual data items that they read. This level of isolation provides all the guarantees of the read committed level. It also ensures that data seen by a transaction does not change until the end of the transaction: i.e., it provides repeatable reads. However, because locks are held on individual data items, transactions may experience the *phantom problem*. If a transaction reads *twice* a set of tuples that satisfy a predicate, it only locks the individual data items that match the predicate. If another transaction inserts a tuple that matches the predicate between the two read operations, that new tuple will appear as a result of the second read.
- **Serializable:** Transactions must acquire long duration read locks on *predicates* as well as on individual data items. This level of isolation protects against all the problems of the less restrictive levels. It ensures serializability.

Name:_____

5. (**20** points) **Transactions: recovery**

   (a) (**2** points) What are STEAL and NO-STEAL policies?
   Solution:

   - **STEAL:** Non-committed updates can overwrite committed values on disk. Requires undo on recovery.
   - **NO-STEAL:** Non-committed updates cannot overwrite committed values on disk.

   (b) (**2** points) What are FORCE and NO-FORCE policies?
   Solution:

   - **FORCE:** All updates made by a transaction must be written to disk before the transaction is allowed to commit.
   - **NO-FORCE:** Updates made by a transaction are not forced to disk before the transaction commits. The buffer manager may write updates to disk after the transaction committed. Requires redo on recovery.

   (c) (**6** points) In ARIES, what is the goal of the analysis phase? Please describe what happens during that phase.
   Solution:
   The purpose of the analysis phase is to determine:

   i. The set of all active transactions at the moment of the crash. These are the transactions that will have to be aborted and their updates undone.
   ii. A conservative superset of all dirty pages at the moment of the crash.
   iii. The point in the log where to start the redo phase.

   To achieve the above goals, the recovery process scans the log from the last checkpoint to the moment when the database crashed. As it does so, it rebuilds the internal data structures: the Dirty Page Table and the Transaction Table. At the end of the phase, the transactions that appear in the Transaction Table are the ones that were active when the failure occurred. The pages that appear in the Dirty Page Table are the conservative estimate of all possibly dirty pages at the moment of the crash. The earliest recoveryLSN in the Dirty Page Table marks the point in the log where redo must start.

Name:_____

(d) (**5** points) In ARIES, what is the goal of the redo phase? Please describe what happens during that phase.

Solution:

The purpose of the redo phase is to rebuild the state of the database at the moment of the failure. This redo paradigm is called *repeating-history*, because ALL updates will be redone, even those made by transactions that will be aborted.

The redo phase start from the point in the log identified by the analysis phase. It scans the log forward. For each update operation found in the log, the recovery process checks that (a) the corresponding page is in the dirty page table, (b) that the page's recoveryLSN is less than or equal to the LSN of the log record, and (c) that the pageLSN of the corresponding page on disk is less than the LSN of the log record. If all three conditions hold, the update is applied.

(e) (**5** points) In ARIES, what is the goal of the undo phase? Please describe what happens during that phase.

Solution:

Once the redo phase finishes, the database is guaranteed to be in the state it was at the moment of the crash. The purpose of the undo phase is to revert all non-committed updates at the moment of the crash to their previous committed states. I.e., all active transactions at the moment of the crash must be aborted and all their updates must be undone. These transactions are called loser transactions.

The undo process is done in reverse chronological order starting from the end of the log (i.e., from the most recent LSN of all transactions to undo). The undo process continues until all updates performed by loser transactions are undone. All undo operations are logged with a special Compensation Log Record (CLR). CLRs ensure that the system never has to undo the effects of an undo.

Name:_____

6. (**12** points) **Distribution**

Please use the description of 2PC from Mohan et. al. to answer the questions below.

(a) (**6** points) In the two-phase commit protocol, what happens if the coordinator sends PREPARE messages and crashes before receiving any votes?

i. (**2** points) What is the sequence of operations at the coordinator after it recovers.

Solution:
After the coordinator restarts, the recovery process will find that a transaction was executing at the time of the crash and that no commit protocol log record had been written (remember that the coordinator does not force-write any log records before sending PREPARE messages). The recovery process will abort the transaction by "undoing" its actions, if any, using the UNDO log records, writing an abort record, and "forgetting" it.

ii. (**2** points) What is the sequence of operations at a subordinate that received the message and replied to it before the coordinator crashed.

Solution:
If the subordinate sent a NO vote, it knows that the transaction will be aborted because a NO vote acts like a veto. The subordinate does not care if the coordinator crashes or not. It aborts the local effects of the transaction and "forgets" about it.

If the subordinate sent a YET vote, it cannot make any unilateral decisions. If the subordinate notices the failure of the coordinator (for example by using a timeout), it hands the transaction over to the recovery process. The recovery process will find that it is in the prepared state for the transaction. It will periodically try to contact the coordinator site to find out how the transaction should be resolved. As we discussed above, after the coordinator recovers, it will abort the transaction and will answer "abort" upon receiving an inquiry message. The subordinate will then abort the transaction and "forget" about it.

iii. (**2** points) What is the sequence of operations at a subordinate that did not receive the message before the coordinator crashed.

Solution:
If the subordinate notices the failure of the coordinator (for example by using a timeout), it hands the transaction over to the recovery process. This time, the recovery process will find no commit protocol log records for this transaction. It will abort it and "forget" about it.

Name:_____

(b) (**6** points) In the two-phase commit protocol, why do subordinates need to force-write a prepared log record before sending a YES VOTE? To answer this question, use an example failure scenario. Show what happens if a subordinate does NOT force-write the prepared log record, then show what happens if the subordinate does force-write the prepared log record.

Solution:
Let's assume that all nodes respond to the coordinator with a YES VOTE. In this case, the coordinator will force-write a commit log record and will send COMMIT messages to all the subordinates. At this point, the transaction is considered to have committed. It should thus be durable.

Let's first consider the case when a subordinate does not force write a prepared log record and crashes after sending a YES VOTE. In this scenario, upon restarting, the recovery process will find no commit log records for the transaction. It will abort the transaction and "forget" about it. The system will then be in an inconsistent state. The transaction should have committed at all sites. Instead, one site aborted it.

Let's now consider the case when a subordinate does force write a prepared log record and crashes after sending a YES VOTE. In this scenario, upon restarting, the recovery process will find that it is in the prepared state for the transaction. It will periodically try to contact the coordinator site to find out how the transaction should be resolved. In our scenario, the final outcome of the transaction is a commit. Because the coordinator cannot forget about a committed transaction until it receives final ACKS from all nodes, it will correctly respond with a COMMIT message to the inquiry. The subordinate will then be able to properly commit the transaction.

Name:_____

7. (**8** points) **Replication**

   (a) (**3** points) In eager master replication, when the master fails, why does a group of replicas need to have the majority of nodes in order to elect a primary and continue processing requests?

   Solution:
   The secondaries cannot distinguish between the crash-failure of the master and a network partition. To maintain consistency, the system must ensure that only one partition processes requests at any time. To ensure this property, only the partition that has the majority of nodes is allowed to elect a new primary and continue.

   (b) (**5** points) What are the differences between eager and lazy replication? Please list differences in the approaches and differences in the properties that result. Discuss master vs group replication if appropriate.

   Solution:
   In eager replication, all updates are applied to all replicas as part of a single transaction. In case of a network partition, only the majority partition is allowed to continue processing requests. This scheme thus favors consistency over availability and has a high runtime overhead.

   In lazy replication, only one replica is updated as part of the original transaction, leading to a better performance than eager replication. Updates then propagate to other replicas asynchronously. With lazy master replication, when the master fails, it is possible for the system to lose the most recent updates that did not yet propagate to other replicas. With lazy group replication, multiple replicas can perform conflicting update operations. Conflicts must later be resolved manually or by using some pre-defined rules. Lazy group replication thus favors availability over consistency. It does not ensure single node serializability. It can only guarantee convergence.