I. Query Execution / Optimization (Derek) [16 pts]

[2 pts]

1. (499^2) * 500 (or 124500500) pages

[5 pts]

2a. 15000 I/Os

1 pt partial credit for 33000 I/Os (did not apply selectivity predicate)

2b. 5000 tuples

[2 pts]

3a. #2

3b. #2

3c. #2

3d. #1

½ pt partial credit per correct choice

[3 pts]

4. a., b.

1 pt partial credit per correct marking

[4 pts]

5. a,b

½ pt partial credit per correct marking

II. Indexing/Storage (Jay) [17 pts]

[5pts]

1a. False

1b. False

1c. True

1d. False

1e. True

[2 point per answer] = 12 pts

Replacement Policy

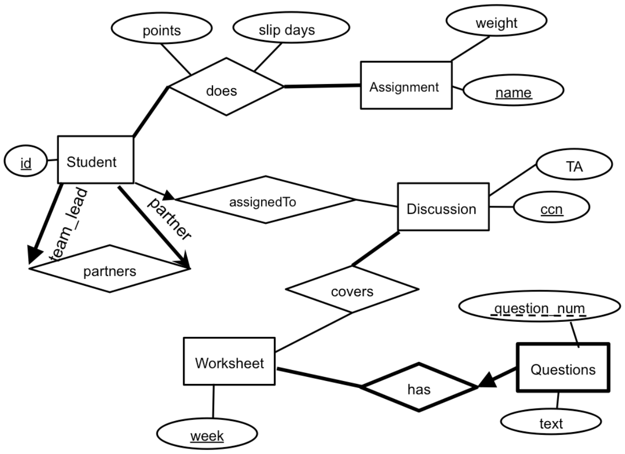
- S1: B - MRU since sequential flooding
- S2: A - LRU for temporal locality
- S3: D - if queries requests are random, then any of the policies will perform approximately equivalently

File Layout, Index:

- S1: D - SortedFile on date, no index

    - just do sequential scans on sorted file
- S2: B - HeapFile with Clustered B+ tree on userid
    - since this constitutes 60% of the data, and fairly write heavy, clustered B+ tree is better choice than SortedFile
    - partial credit for SortedFile on userid, No index
- S3: A - heapfile, Unclustered B+ tree on postid
    - use unclustered for point lookups, and no need for sorted file

III. Database Design (Michelle) [18 pts]



1.
   3.5 points: ½ pt for each edge, underlined key
2. Questions, see diagram above for notation
   1.5 points: .5 for partial key, .5 for bolded entity, .5 for bolded reln
3. Relation from student to itself or using aggregation
   2 points: 1 pts for making a new entity and a relationship
4. a. A; b. E; c. NMI; d: E
5. iv
   2 points: .5 for each
6. a. NW, INS, NQTPR b. Yes; 3 pts
7. INQ -> RP, (or INQ -> R, INQ -> P), NQ -> T, NQR -> P; 4 pts

IV. Concurrency (Vikram) [11.5pts]

1. 1 point each
    a. False
    b. False
    c. True
    d. False

        e. True
2. 1 point each
    a. Yes
    b. No
3. T2, T3, T1, T4: 1 point each
4. i, l

V. Recovery (Varun) [9 pts]

1.

LSN	Record	prevLSN
80	CLR: T1 LSN 0	70

2.

XID	Status	lastLSN
T3	Running	100
T2	Aborting	120

For partial credit, we accepted Aborting instead of Running for T3, because all transactions are essentially aborting during the UNDO phase.

3.

PID	recLSN
P2	20
P3	40
P4	50
P1	100

4. 20, 40, 50, 70, 80, 100, 120

5.

LSN	Record	prevLSN
200	CLR: T3 LSN 100	100
210	END: T3	200
220	CLR: T2 LSN 40	120
230	END: T2	220

6. a, b. a is correct, because during the REDO phase of recovery, some UPDATE log records that reflect writes that never made it to disk will be skipped. Similarly, b is correct, because some CLR's that reflect UNDO's that never made it to disk will be skipped. c is incorrect because no COMMIT log records are written during recovery. d is incorrect because even if REDO begins at a later LSN, the system does not add any new transactions to the transaction table during REDO.

VI. SQL Anthony [15 pts]

```
Passenger(
     pid (int),
     first_name (text) NOT NULL,
     last_name (text) NOT NULL
)

Driver(
     did (int),
     first_name (text) NOT NULL,
     last_name (text) NOT NULL
)

Trip(
     tid (int),
     pid (int references Passenger(pid) NOT NULL),
     did (int references Driver(did) NOT NULL),
     start_time (timestamp) NOT NULL,
     end_time (timestamp) NOT NULL,
     distance (decimal) NOT NULL,
     passenger_rating (decimal),
     driver_rating (decimal)
)
```

1. You hypothesize that some months of the year are more popular than others, perhaps due to weather or special events like holidays. To assess this, you want to know how many trips were completed in each month, independent of year.

```
CREATE VIEW num_trips_by_month AS
SELECT EXTRACT(MONTH FROM start_time) AS month,
       COUNT(*) AS num_trips
FROM Trips
GROUP BY EXTRACT(MONTH FROM start_time);
```

Note: `EXTRACT(MONTH FROM _____)` is a PostgreSQL function that extracts the numeric month (e.g. January = 1) out of a timestamp or interval.

2. You want to prepare your staff next year to improve heavily on the poorest performing month(s) (independent of year). Which month(s) had the minimum number of trips? (Use the view created in Q1)

```
SELECT  month
FROM  num_trips_by_month NTM JOIN
      (
           SELECT MIN(NTM_TMP.num_trips) AS min_column
           FROM  num_trips_by_month NTM_TMP;
      ) MIN_TABLE
      ON NTM.num_trips = MIN_TABLE.min_column;
```

3. a. Which drivers have a perfect 5.0 average rating from all their trips that received driver ratings? (Return just the unique `did`)

```
SELECT  T1.did
FROM  Trip AS T1
WHERE  5.0 = ALL(
      SELECT T2.driver_rating
      FROM  Trip AS T2
      WHERE  T1.did = T2.did AND driver_rating <> NULL;
);
```

also

```
SELECT  T1.did
FROM  Trip AS T1
WHERE  NOT EXISTS (
      SELECT *
      FROM  Trip AS T2
      WHERE  T1.did = T2.did AND driver_rating < 5.0;
```

b. When executing this query, you find that this query runs very slowly. What about the structure of this query may cause it to execute so slowly?

A. B. C.

c. You attempt re-writing this same query with the hope of speeding it up.

```
SELECT  T.did
FROM  Trip as T
GROUP BY  T.did
HAVING [AVG|MIN](T.driver_rating) = 5.0;
```

4. You hypothesize that drivers and passengers with the same first name get along better (have better ratings) than drivers and passengers that don’t share any commonalities.

Notice that the passenger and driver ratings can be NULL.

Select the queries that yield the desired result.

```
SELECT SAME_NAME.rating AS same_name,
       DIFF_NAME.rating AS diff_name
FROM  (SELECT AVG(driver_rating) AS rating
       FROM  Passenger P, Driver D, Trip T
       WHERE  P.pid = T.pid AND D.did = T.did
              AND P.first_name = D.first_name) AS SAME_NAME,
      (SELECT AVG(driver_rating) AS rating
       FROM  Passenger P, Driver D, Trip T
       WHERE  P.pid = T.pid AND D.did = T.did
             AND P.first_name <> D.first_name) AS DIFF_NAME;
```

```
SELECT SAME_NAME.rating AS same_name,
       DIFF_NAME.rating AS diff_name
FROM  (SELECT (SUM(driver_rating) / COUNT(*)) AS rating
       FROM  Passenger P, Driver D, Trip T
       WHERE  P.pid = T.pid AND D.did = T.did
              AND P.first_name = D.first_name) AS SAME_NAME,
      (SELECT (SUM(driver_rating) / COUNT(*)) AS rating
       FROM  Passenger P, Driver D, Trip T
       WHERE  P.pid = T.pid AND D.did = T.did
             AND P.first_name <> D.first_name) AS DIFF_NAME;
```

```
SELECT  AVG(TS.driver_rating) AS same_name,
        AVG(TD.driver_rating) AS diff_name
FROM  Passenger PS, Driver DS, Trip TS,
      Passenger PD, Driver DD, Trip TD
WHERE PS.pid = TS.pid AND DS.did = TS.did AND
      PD.pid = TD.pid AND DD.did = TD.did AND
      PS.first_name = DS.first_name AND
      PD.first_name <> DD.first_name;
```

anthonysutardja 5/10/15 3:04 PM

Comment: tested http://sqlfiddle.com/#!15/75a7b/1