Example Queries

- Find the names of all instructors whose department is in the Watson building

$$\{t \mid \exists s \in \textit{instructor}\ (t[\textit{name}] = s[\textit{name}] \wedge \exists u \in \textit{department}\ (u[\textit{dept\_name}] = s[\textit{dept\_name}] \wedge u[\textit{building}] = \text{“Watson”}))\}$$

- Find the set of all courses taught in the Fall 2009 semester, or in the Spring 2010 semester, or both

$$\{t \mid \exists s \in \textit{section}\ (t[\textit{course\_id}] = s[\textit{course\_id}] \wedge s[\textit{semester}] = \text{“Fall”} \wedge s[\text{year}] = 2009) \vee \exists u \in \textit{section}\ (t[\textit{course\_id}] = u[\textit{course\_id}] \wedge u[\textit{semester}] = \text{“Spring”} \wedge u[\text{year}] = 2010)\}$$

Universal Quantification

- Find all students who have taken all courses offered in the Biology department
  - $\{t \mid \exists\, r \in \mathit{student}\ (t\,[\mathit{ID}] = r\,[\mathit{ID}]) \wedge$
    $(\forall\, u \in \mathit{course}\ (u\,[\mathit{dept\_name}] = \text{“Biology”} \Rightarrow$
    $\exists\, s \in \mathit{takes}\ (t\,[\mathit{ID}] = s\,[\mathit{ID}\,] \wedge$
    $s\,[\mathit{course\_id}] = u\,[\mathit{course\_id}]))\}$
  - Note that without the existential quantification on student, the above query would be unsafe if the Biology department has not offered any courses.

Case Statement for Conditional Updates

- Same query as before but with case statement

```
update instructor
    set salary = case
                    when salary <= 100000 then salary * 1.05
                    else salary * 1.03
                 end
```

Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all accounts owned by a given customer

```
create function instructors_of (dept_name char(20)
        returns table (  ID varchar(5),
                         name varchar(20),
                            dept_name varchar(20),
                         salary numeric(8,2))
return table
        (select ID, name, dept_name, salary
         from instructor
         where instructor.dept_name = instructors_of.dept_name)
```

- Usage

```
select *
from table (instructors_of ( 'Music' ))
```

Trigger to Maintain credits_earned value

- **create trigger** *credits_earned* **after update of** *takes* **on** (*grade*)
  referencing new row as *nrow*
  referencing old row as *orow*
  for each row
  when *nrow.grade* <> 'F' **and** *nrow.grade* **is not null**
  and (*orow.grade* = 'F' **or** *orow.grade* **is null**)
  begin atomic
  update *student*
  set *tot_cred*= *tot_cred* +
  (**select** *credits*
  from *course*
  where *course.course_id*= *nrow.course_id*)
  where *student.id* = *nrow.id*;
  end;

```
FROM LandlordofResidence
GROUP BY lid,lcity
)
AS temp1,
(SELECT a.lcity, MAX(countrid)as maxcount
FROM (SELECT lcity,lid,COUNT(DISTINCT rid) as countrid
FROM LandlordofResidence
GROUP BY lid,lcity) as a
GROUP BY lcity
)
AS temp2
WHERE temp1.lcity = temp2.lcity AND temp1.countrid = temp2.maxcount
```

(ii) Change the name of any residence with name “Little Cabin” to “Big Cabin”.

```
UPDATE LandlordofResidence
SET rname = 'Big Cabin'
WHERE rname = 'Small Cabin'
```

(b)

If you are using mysql,Then the trigger cannot be implemented because "A stored function or trigger cannot modify a table that is already being used (for reading or writing) by the statement that invoked the function or trigger." according to https://dev.mysql.com/doc/refman/5.7/en/stored-program-restrictions.html#stored-routines-function-restrictions

```
CREATE trigger hw2b
AFTER INSERT on Rating
FOR EACH ROW
IF
((SELECT AVG(score) as avg_score
FROM Rating r
WHERE r.rid = NEW.rid) <3
AND EXISTS (SELECT 1 From Rating r2 WHERE score = '1' AND
r2.rid = NEW.rid))
THEN
DELETE FROM Rating
WHERE score = '1' AND Rating.rid = NEW.rid;
END IF
```

(c)

```
CREATE trigger hw2c
Before INSERT on Rating
FOR EACH ROW
IF
(
SELECT count (*)
FROM Rating
WHERE NEW.cid = cid AND score = '1' and timestampdiff(Week, rtime,
CURDATE())<=2
```

NYU School of Engineering February 4, 2018
Computer Science and Engineering
CS 6083, Spring 2018

Problem Set #1 (due 2/15)

Note: In this homework, you do not need to create tables and execute queries using an actual DBMS. A written solution is sufficient. Also, while the problems talk about relational schemas for websites, you do not have to worry about how to build such a site, but only need to think about suitable underlying schemas.

Problem 1: Suppose you have a database modeling a system for users to review and rate restaurants. It is given by the following highly simplified schema:

USER(<u>uid</u>, uname, uphone, ucity, ustate)

RESTAURANT(<u>rid</u>, rname, raddress, rcity, rstate, rstar, ropen)

REVIEW(<u>reviewid</u>, rid, uid, text, rdate)

RATE(<u>rid, uid</u>, rate)

FRIEND(<u>uid, fid</u>)

In this schema, uid is the unique ID of a user and rid is the unique ID of a restaurant, rstar is the average rating between 1 and 5 given by users for a restaurant, and ropen indicates whether the restaurant is still open for business. Because users may visit the same restaurant several times, and each time they might have a different experience, a single user may write several reviews for the same restaurant. Reviewid is the unique ID of a review. Users can also rate restaurants. However, for each user and restaurant there is only one rating -- a user might later update their rating, but only the last rating is stored. Also, users can make friends with each other.

(a) Identify suitable foreign keys for this schema.

(b) Suppose we do not have the attribute “reviewid” in the REVIEW table. How would you choose a suitable primary key in this case?

(c) Write statements in SQL for the following queries.

I. List the names of all restaurants in Brooklyn that have never received a rating of five stars.

II. Output the uid and uname for any user living in Seattle who gave a restaurant named “Good BBQ” a five-star rating.

III. Output the name of any restaurant that has received at least two one-star ratings.

IV. Output the uid and uname of any user who has written reviews for every restaurant in Ohio.

V. Output the name of any user that gave a restaurant a rating of five stars, and where at least ten friends of that user also gave the same restaurant five stars.

VI. For each restaurant, output the rid, and the uid and uname of the user who wrote the most recent review for that restaurant.

VII. We say that a rating of a restaurant is local if it is given by a user from the same city as the restaurant. For each city, output the restaurant with the highest average local rating.

(d) Write expressions in Relational Algebra for the above queries.

(e) Write either DRC or TRC queries for the above queries. Or explain the reason why you think a particular query cannot be done in DRC or TRC (or would be extremely complicated to do).

Problem 2: In this problem, you need to design a relational schema for a coupon deal shopping website. This is a website that includes many different merchants, and allows customers to order from these merchants and to apply suitable coupons offered by the website.

For each mechant, you need to store detailed information, including a unique id, its name, and its phone number. For each products, you need to store a unique id, a name, a product category (e.g., "Tools", or "Toys"), a brand name (e.g., "Samsung"), and a brief description. Notice that the same product could be sold by several merchants at different prices, so you need to store which merchants offer a product at what price. For each customer, you need to store a unique id, name, phone number, and a shipping address. Each customer may have several credit cards having the same or different billing addresses, but each credit card can belong to only one customer. While shopping, customers may place items into a shopping card, and you need to keep track of items currently in the cart, their quantities, and their prices. Note that there is a different shopping cart for each merchant.

Coupon deal could be quite complicated, say "Buy 2 and get the 3rd one free", or "Buy any vacuum cleaner and get a free toaster worth up to $20". To simplify things, we assume that coupon deals always have the following format: A coupon deal has a unique deal ID, a minimum purchase amount, a percentage discount, a product category, a brand name, and an expiration date. For example, a coupon deal might offer 20% off on a purchase of at least $100, say for any phones made by Samsung (category = phones, brand = Samsung), or for any products made by Samsung (category = ALL, brand = Samsung), for any cell phones (category = phones, brand = ALL), or any products (category = ALL, brand = ALL). For each coupon deal, each merchant can decide whether or not to participate in this deal, and you need to store this information.

We assume that all coupon deals are available to all customers, and that a customer may use the same coupon several times. (So deals are not personalized for particular customers, and there is no limit on the number of deals a customer can participate in.) However, when a customer makes a purchase, only one coupon deal can be applied to this purchase, and the purchase can only contain (one or more) items from a single merchant. Thus, a customer cannot combine products from several merchants in order to reach the minimum purchase amount required for the coupon deal, even if the products the customer wants cannot all be bought from the same merchant. For each order, we need to store the credit card used, the items ordered and their quantities and prices, the coupon that was used (if any), the total price before and after applying the coupon, plus the date and time of the order. For simplicity, we assume there are no taxes and no shipping charges.

(e) For 5,7, we need to use count or average, which are not applicable in RC.

1. $\{t \mid \exists s \in RESTAURANT(t[rid] = s[rid] \wedge t[rname] = s[rname] \wedge s[rstate] = 'Brooklyn') \wedge \neg \exists u \in RATE(t[rid] = u[rid] \wedge u[rate] = 5)\}$

2. $\{t \mid \exists s \in USER(t[uid] = s[uid] \wedge t[uname] = s[uname] \wedge s[ucity] = 'Seattle' \wedge \exists u \in RATE(s[uid] = u[uid] \wedge u[rate] = 5 \wedge \exists r \in RESTAURANT(u[rid] = r[rid] \wedge r[rname] = 'Good\ BBQ')))\}$

3. $\{t \mid s \in RESTAURANT(t[rname] = s[rname] \wedge \exists r \in RATE(r[rid] = s[rid] \wedge r[rate] = 1 \wedge \exists r_1 \in RATE(r_1[uid] <> r[uid] \wedge r_1[rate] = 1)))\}$

4. $\{t \mid \exists s \in USER(t[uid] = s[uid]) \wedge (\forall u \in RESTAURANT(u[rstate] = 'Ohio' \Rightarrow \exists r \in REVIEW(t[uid] = r[uid] \wedge r[rid] = u[uid]))\}$

6. $\{t \mid \exists u \in USER(t[uid] = u[uid] \wedge t[uname] = u[uname] \wedge \exists s \in REVIEW(u[uid] = s[uid] \wedge t[rid] = s[rid] \wedge (\forall s_1 \in REVIEW(s_1[rid] = s[rid] \Rightarrow s[rdate] >= s_1[rdate])))))\}$

Problem 2

(a)

Customer(**cid**, cname, cphone, shipaddress, ccity, cstate, ccountry)

Merchant(**mid**, mname, mphone)

Product(**pid**, pname, pcategory, pbrand, pdesc)

NYU Tandon School of Engineering **Nov 30, 2017**
Computer Science and Engineering
CS6083, Fall 2017

Problem Set 5

Problem 1

Consider the three schedules below. For each one, answer the following questions and provide detailed explanation or proof for your answer:
Note: "is this possible under a strict 2PL protocol" means does there exist a way to lock the items as needed, so that each transactions goes through a growing and a shrinking phase.

a) Is this schedule conflict-serializable?
b) Is this schedule recoverable if the transactions complete in the order specified?
c) Is this schedule cascadeless?
d) Is this schedule possible under a (non-strict) 2PL protocol?
e) Is this schedule possible under a strict 2PL protocol?
f) Is this schedule possible under a rigorous 2PL protocol
g) For the third schedule, if it's not in 2PL protocol, give a possible answer so it can be 2PL.

T1	T2	T3
R(A)		
	R(B)	
W(C)		
		R(C)
	W(B)	
W(A)		
		R(B)
R(B)	abort	
commit		W(C)
		commit

Table 1: Schedule 1

T1	T2
	R(A)
R(B)	
R(A)	
W(A)	
	R(C)
W(B)	
commit	W(C)
	commit

Table 2 : Schedule 2

T1	T2	T3
	R(A)	
R(B)		
		R(C)
R(A)		
		R(B)
W(A)		
	R(C)	
		W(C)
W(B)		
commit		
	W(C)	
	commit	
		R(A)
		commit

Table 3: Schedule 3

Problem 2

Consider the following schedule on five transactions, where lock-X() and lock-S() denote requests for exclusive and shared locks, respectively, Is the schedule deadlocked? Prove your answer. If it is deadlocked discuss one method to recover. If it is not deadlocked, how would you avoid a deadlock?

T1	T2	T3	T4	T5
lock-S(A)				
		lock-S(C)		
	lock-X(B)			
			lock-X(D)	
				lock-S(E)
lock-S(E)				
		lock-S(G)		
	lock-S(A)			
				lock-X(C)
			lock-S(C)	
	lock-X(G)			
lock-S(D)				
				lock-S(B)
			lock-X(B)	
		lock-S(A)		

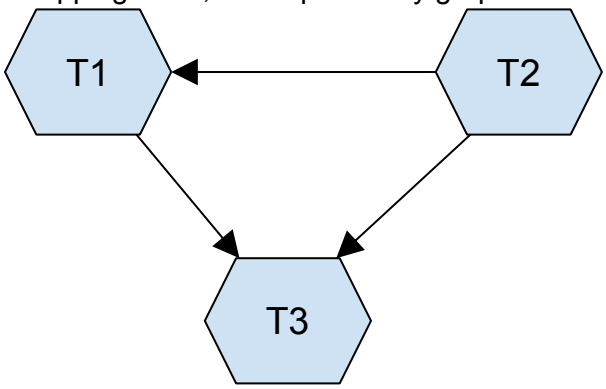
NYU Tandon School of Engineering **Nov 30, 2017**
Computer Science and Engineering
CS6083, Fall 2017

Sample Solution 5

Problem 1

Schedule 1

(a) Yes, the equivalent serial schedule is T2 -> T1 -> T3 which can be achieved through swapping. Also, the dependency graph reveals no circles:



Dependency graph for Schedule 1

(b) No. T3 has read a value of 'B' that has been written by T2. When T2 aborts, this value must be rolled back, but T3 has already committed, so this is not possible.

(c) No. Both T1 and T3 have read B value written by T2. Failure of T2 will cause both to be rolled back (aka cascading rollback). If the commit operation of T2 appears before the read operation of T1 and T3, it's cascadeless schedules.

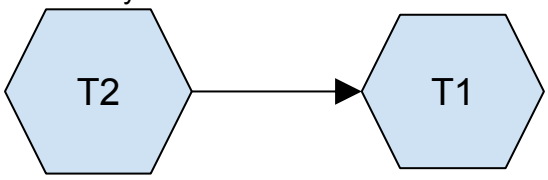
(d) No. T3 would not be able to obtain a lock for reading the value of 'C', because T1 has already acquired a lock (Exclusive) on T1 and after writing on C, T1 will write on A and read B, so it's still in growing phase and cannot release any lock.

(e) No. The schedule is not two-phase (aka it is not cascadeless)

(f) No. Same reason

Schedule 2

(a) Yes. The equivalent serial schedule is T2 -> T1. The dependency graph does not contain cycles:



(b) Yes. Neither of the transaction reads a value written previously by the other one.

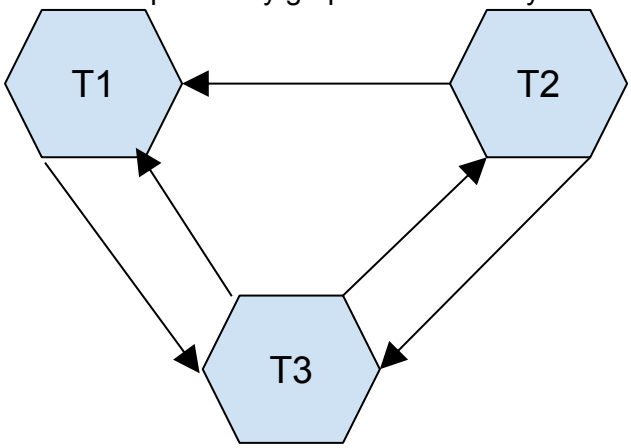
(c) Yes. Same reason as(b)

(d) No. T1 cannot acquire an exclusive lock on 'A' due to T2 already having a shared lock on 'A', which it cannot release yet since it needs to acquire additional locks laer.

(e) No. Due to (d)
(f) No. Due to (d)

Schedule 3

(a) No. The dependency graph contains a cycle



(b) Yes, only T3 reads value of ‘A’ written by another transaction T1 and T1 commits before T3
(c) Yes. If we assume that the R(A) of T3 happens after the commit of T2. No, otherwise
(d) No. T1 cannot acquire an exclusive lock on ‘A’ due to T2’s shared lock on ‘A’
(e) No. Due to (d)
(f) No. Due to (d)
(g)

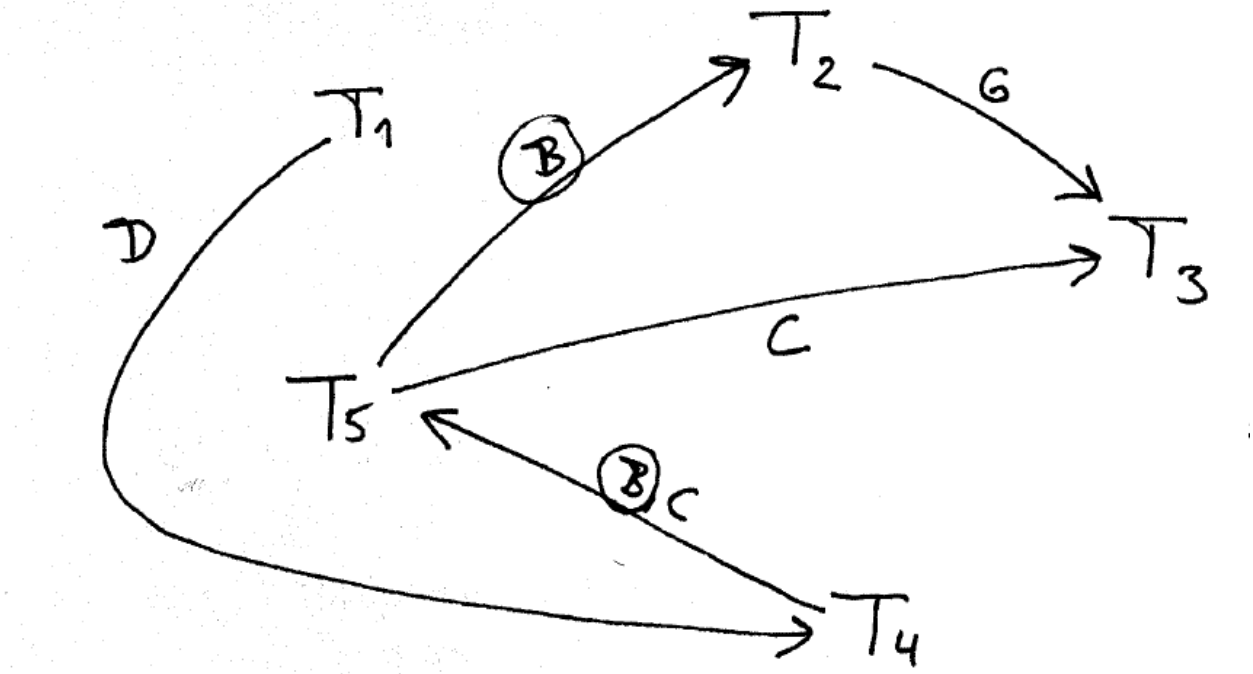
T1	T2	T3
R(B)		
R(A)		
W(A)		
	R(C)	
W(B)		
commit		
	W(C)	
	commit	
		R(A)
		commit

Problem 2

Note: In the following we assume the use of 2PL. This means that nobody can release any locks until they have acquired the last lock. The analysis also holds for strict and rigorous 2PL.

Wait-for graph:
For each wait-for edge, we show the object that is involved. For example, T1 waits for T4 on D.

Note: B is circled because these wait-for relationships would not exist if T2 and T5 release their locks immediately after the last operation on the schedule shown.



These is no cycle, and no deadlock. Transaction could finish in order T3, T2, T5, T4, T1.

F = {BF -> C, CF -> D, G -> A, G->E, FG -> AD}

1. Derive all candidate keys for this schema.
2. Derive a canonical cover of the functional dependencies in F
3. Is the above schema in BCNF? Prove or disprove. If it is not in BCNF, convert it into BCNF.
4. In the BCNF schema from c) dependency-preserving? Prove or disprove. If not, convert it into 3NF.

Problem2.

(a) Candidate key: {BFG}

(b) Canonicalcover:Fc={BF→C,CF→D,G→AE,FG→D}.

(c) It's not in BCNF because there is nontrivial functional dependency BF → C, and BF is not superkey.

Decomposing relation R into BCNF:

Because BF → C : R1 = (B, C, F), and R2 = (A, B, D, E, F, G).

Because G → AE, and G is not superkey, decomposing relation R2 into BCNF: R2 = (A, E, G), and R3 = (B, D, F, G).

Because F G → D, and F G is not superkey, decomposing relation R3 into BCNF: R3 = (D, F, G), and R4 = (B, F, G). And the BCNF form is: R1 = (B, C, F), R2 = (A, E, G), R3 = (D, F, G), and R4 = (B, F, G).

(d) No. The BCNF form in (c) is not dependency preserving. We cannot check CF → D in the result of (c).

3NF form: R1 = (B, C, F), R2 = (C, D, F), R3 = (A, E, G), R4 = (D, F, G), and R5 = (B, F, G).

F = {DF->A, E->A, FB->C, D->A, E->G, F->BG}
So the canonical cover will be: Fc = { D→A, E→AG, F→BCG}
Hence BCNF form is: R1 = {A, D} R2 = {B, C, F, G} R3 = {D, E, F}
Therefore, 3NF form is: R1 = {A, D} R2 = {A, E, G} R3 = {B, C, F, G} and R4 = {D, E, F}

c.

```
SELECT table_name, count(column_name) as colcount
FROM information_schema.columns natural join information_schema.tables
WHERE TABLE_SCHEMA = 'hw1'
GROUP BY TABLE_NAME;

SELECT table_name,column_name
FROM information_schema.columns natural join information_schema.tables
WHERE table_schema = 'hw1' and DATA_TYPE = 'int';
```

Unclustered index on aid in Artist table:

Each tree node contains n-1 keys and n pointers. With 8 bytes per key, 12 bytes per pointer and a node size of 4096 bytes, we can find how many keys and pointers can fit in each node: (n-1)*8 + 12*n = 4096 => n = 205. Assuming 80% occupancy per node, each leaf node will contain about 164 index entries and each internal node will have 164 children.

An unclustered index on the Artist table will have one index entry for each record, or a total of 1M index entries. Since about 164 index entries are in each leaf node there will be about 1M/164 = 6098 leaf nodes, on the next level we have 37 nodes, then the next would be the root of the tree, so the B+ tree has 3 levels of nodes. It takes about 4*5ms = 20ms to fetch a single record from the table using this index without caching. The size of the tree is (6098 + 37 + 1)*4KB = 24.5MB.

Dense Clustered index on (tid) in Play table:

Each tree node contains n-1 keys and n pointers. With an 8-byte tid key, 12 bytes per pointer and a node size of 4096 bytes. Assuming page size = 4KB(most common page size in various operating systems), since (n - 1) * 8 + 12 *n = 4096, we can get that n is about 205, and with 80% occupancy we get about 164 entries per node.

There are 20 billion records, with 1 record for each entry, we have 20 billion entries in total, and thus there are about 122M nodes at the leaf level, 740K nodes at the next level, then 4.5K, then 25, then the root. The tree has 5 levels of nodes.

We need (5 + 1)*5ms = 30ms to fetch a single record from the table. The seeking time of fetching all records of one particular tid would be also 30ms, because we are using clustered index, all records with the same tid are stored continuously, we do not need to cost extra seek time. The time used to fetch 50 records would be 30ms + 50*40bytes/100(MB/s) ~ 30ms. The size of the tree is (122M + 740K + 4.5K + 25 + 1) * 4KB = 491GB.

(Appendix) Sparse Clustered index on (tid) in Play table:

Each tree node contains n-1 keys and n pointers. With an 8-byte tid key, 12 bytes per pointer and a node size of 4096 bytes. Assuming page size = 4KB(most common page size in various operating systems), since (n - 1) * 8 + 12 *n = 4096, So n is about 205, and with 80% occupancy we get about 164 entries per node.

There are 20 billion records, with 4KB/40bytes = 100 records for each entry, we have 200M entries in total, and thus there are about 1220K nodes at the leaf level, 7.4K nodes at the next level, then 45, then the root, so the tree has 4 levels of nodes.

We need (4 + 1)*5ms = 25ms to fetch a single record from the table. Because we are using clustered index, all records with the same tid are stored continuously, we do not need to cost extra seek time. The time used to fetch 50 records would be 25ms + 50*40bytes/100(MB/s) ~ 25ms. The size of the tree is (1220K + 7.4K + 45 + 1) * 4KB = 4.9GB.

(a) The capacity of this disk is (3*200000*1000*1024) = 600GB

The maximum data transfer speed is 1000*1024*12000/60 = 200MB/s The average rotate latency is (60s*1000)/2/12000 = 2.5ms

(b)Note 200MB/S ~ 200KB/ms, so it takes 1/200KB = 0.005ms to read 1 KB of data

Block model: (with 4KB per block)

Time to read 200K: 50t = 50*6.52=326ms

Time to read 20M: 5000t = 5000*6.52=32.6s Time to read 2G: 500000t = 500000*6.52=3260s

LTR model:

read 200KB: time = 4ms + 2.5ms + 1ms = 7.5ms read 20MB: time = 4ms + 2.5ms + 100ms = 106.5ms read 2GB: time = 4ms + 2.5ms + 10s = 10.0065s

(c)

Phase 1: Repeat the following until all data is read: Read 4GB of data and sort it in main memory using sorting algorithm. Write it into new file until all data is read. The time to read 4 GB is (6.5ms + 20s). To read and write 64 such files takes 2*64*(6.5ms+20s) = 2560s
Phase 2: 64 input buffers and 1 output buffer of 4G/65=61.54MB each. Reading/writing one buffer takes 6.5ms + (61.54/200) = 307ms. Reading all 256 GB in 4260 pieces of 61.54MB takes 307 * 4260/1000 = 1308s. Each pass takes 2616s. Total time = 2616 + 2560 = 5176s
(d)
We now have two 8-way merges. For each merge, we have 8 input buffers and 1 output buffer of about 444.44MB (= 4000/9) each. Reading/writing one buffer takes 6.5ms + 444.44/200 = 2.18s. Reading all 256GB in 590 pieces of 444.44MB takes 590 * 2.18 = 1286s. Thus, each pass takes 2 * 1286= 2572s. Both passes take 5144s. Total time = 5144+2560=7704s.