University of California, Berkeley
College of Engineering
Computer Science Division – EECS

Spring 2011 Prof. Michael J. Franklin

MIDTERM I

CS 186 Introduction to Database Systems

NAME:____________________________ STUDENT ID:__________________

IMPORTANT: Circle the last two letters of your class account:

```
cs186 a b c d e f g h i j k l m n o p q r s t u v w x y z
      a b c d e f g h i j k l m n o p q r s t u v w x y z
```

DISCUSSION SECTION DAY & TIME:______________ TA NAME: _____________

This is a **closed book** examination – but you are allowed one 8.5" x 11" sheet of notes (double sided). You should answer as many questions as possible. Partial credit will be given where appropriate. There are 100 points in all. You should read **all** of the questions before starting the exam, as some of the questions are substantially more time-consuming than others.

Write all of your answers directly on this paper. **Be sure to clearly indicate your final answer** for each question. Also, be sure to state any assumptions that you are making in your answers.

GOOD LUCK!!!

Problem	Possible	Score
1. Buffer Manager and Storage	**26**	
2. B+ Trees	**30**	
3. Hash Indexes	**15**	
4. Transactions and Concurrency Control	**29**	
TOTAL	**100**	

Question 1 – Buffer Management and Storage [5 parts, 26 points total]

For parts (a) to (d), circle your answer and be sure to state why in a sentence or two. List any assumptions you are making.

a) [3 points] Which access pattern would see a greater improvement in performance when upgrading the storage media from hard disk drives to solid state (e.g., flash) drives?

i) Sequential scan **<u>ii) Random access</u>** **iii)** Both improve the same

Why?

For random access, SSD doesn't have to rotate (seek).

b) [3 points] Consider an access pattern where there are lookups of many random keys using a static hash index that has no overflow pages. If the hash index is larger than the buffer pool, which buffer replacement policy will provide the best hit rate?

i) LRU **ii)** MRU **iii)** Random **<u>iv) All are similar</u>**

Why?

For random access without any hierarchy (static hash without no overflow pages), there is no way to predict which page is needed in the future.

c) [3 points] Consider an access pattern where the database does repeated sequential scans of a table (sequential flooding). If the size of the table is **slightly** greater than the buffer pool, which buffer replacement policy will provide a significantly better hit rate than the others?

i) LRU **<u>ii) MRU</u>** **iii)** LRI **iv)** All are similar

Why?

LRU and LRI suffer from sequential flooding (0% hit rate).

d) [3 points] Why is physical data independence useful? Give a simple example.

The logical schema can remain unchanged even though the physical storage changes. Some examples are: Addition or removal of an index, partitioning of table across multiple disks, partitioning of table across multiple machines…

Question 1 – Buffer Management and Storage (continued)

e) [14 points] Consider a database system with 4 buffer frames (#1, #2, #3, #4) and a file of 6 disk pages (A, B, C, D, E, F). Assume that you start with an empty buffer pool. A sequence of requests is made to the buffer manager as described in the Request column (below). At certain times a Pin request is immediately followed by an Unpin request (represented as Pin/Unpin), but other times Pin and Unpin requests happen in an interlaced manner with other requests.

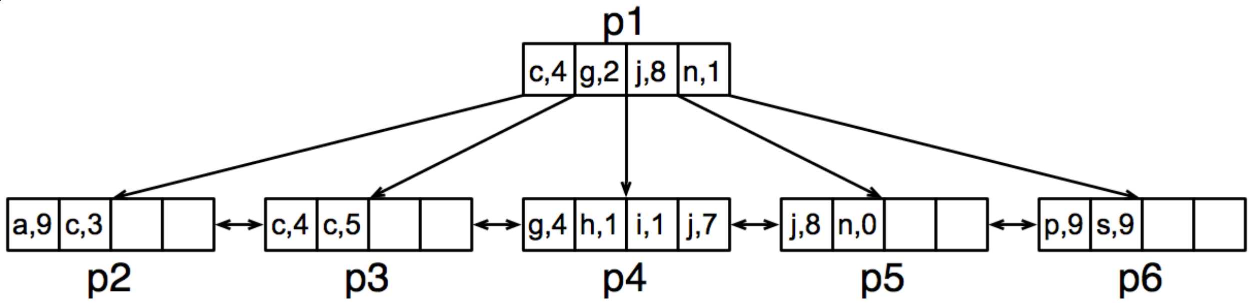
Fill in the following table showing the buffer contents after the completion of each operation using the **CLOCK** page replacement policy, as described in the book and HW 1. For each page, indicate the pin count (PC) and for unpinned pages, indicate the value of the reference bit (1 or 0). You should mark unchanged buffer frames with "ditto" ("). As in HW 1 we make the following assumptions in the clock policy:

The pointer doesn't move when filling a free frame in buffer or on a buffer hit.

The pointer advances after replacing a buffer frame.

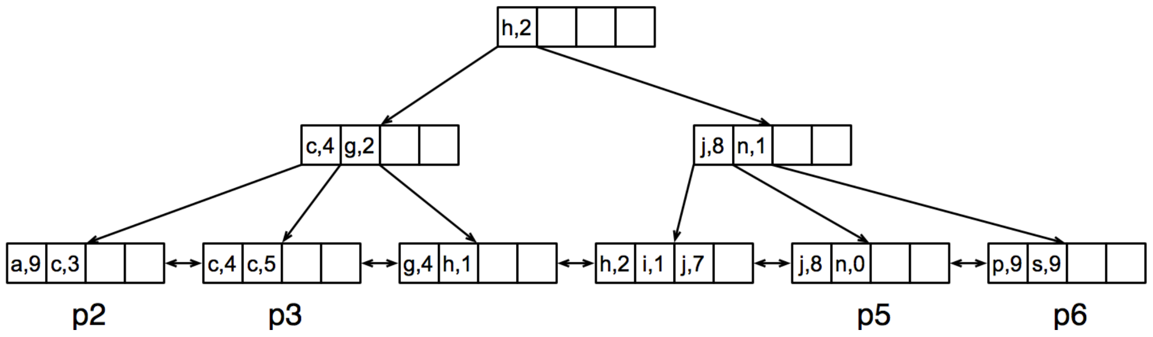
Time	**Request**	**Buffer Frames**			
		#1	**#2**	**#3**	**#4**
T1	Pin A	A PC=1	Empty	Empty	Empty
T2	Pin/Unpin B	"	B PC=0, Ref=1	"	"
T3	Pin/Unpin C	"	"	C PC=0, Ref=1	"
T4	Pin/Unpin D	"	"	"	D PC=0, Ref=1
T5	Pin A	A PC=2	"	"	"
T6	Unpin A	A PC=1	"	"	"
T7	Pin/Unpin B	"	"	"	"
T8	Pin/Unpin E	"	E PC=0, Ref=1	C PC=0, Ref=0	D PC=0, Ref=0
T9	Pin/Unpin F	"	"	F PC=0, Ref=1	"
T10	Pin/Unpin B	"	"	"	B PC=0, Ref=1
T11	Pin F	"	"	F PC=1	"

Question 2 – B+ Trees [7 parts, 30 points total]
For parts (**a)-(c**), consider the following B+ tree with order d=2 and a composite search key and use the B+tree algorithms discussed in class and in the book

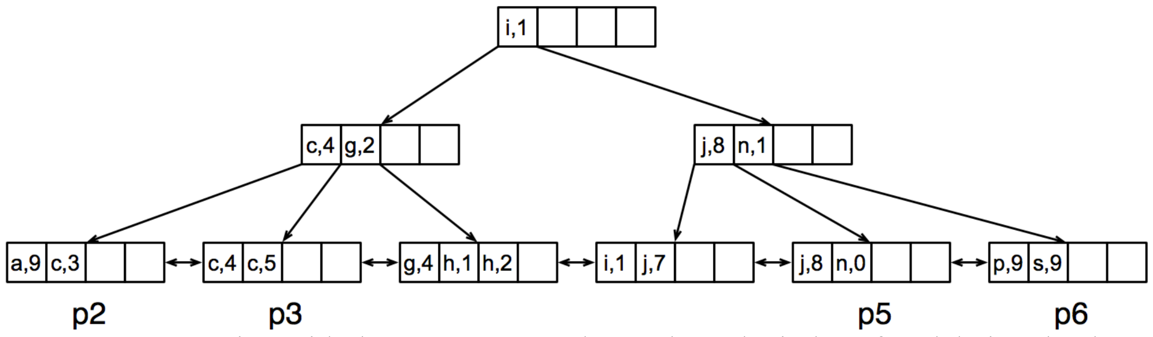


a) [2 points] What is the maximum number of keys you could insert that would NOT change the height of the tree?
8 keys

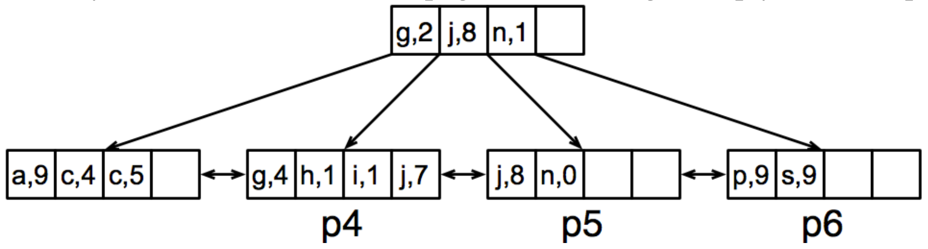
b) [5 points] For the **original** tree above, draw the index after inserting a data entry with key **h,2**. If the contents of a page do not change, please write the page label (e.g,, p1, p2, …) rather than drawing the page.



or



c) [5 points] Starting with the **original** tree above, draw the index after deleting the data entry with key **c,3**. If the contents of a page do not change, simply write the page label.



Question 2 – B+ Trees (continued)

For parts **(d) – (g),** consider the following scenario: The latest hot Internet start up, Pawbook (a social network for pets), is having performance problems and they suspect the cause is their database system. They hire you to help. The Pawbook database has a USERS table containing a record for each user. When someone logs into Pawbook, an equality lookup query is issued on the USERS table to find the record associated with the USER_ID. The USERS table is organized as a sequentially allocated **sorted file**, sorted on USER_ID, and there are no indexes.

The system has the following parameters (note – the fact that all values are powers of 2 should make your calculations easy, and you can report answers using powers of 2 as well):

The USERS table contains 4194304 (2^{22}) records

A USERS data record is 256 (2^8) bytes long

Each page is 16384 (2^{14}) bytes, and we ignore any page header overhead so 64 (2^6) USERS records can fit in each page

There is only one disk and it can support up to 256 (2^8) random IOs per second

d) [5 points] You notice that there are no indexes on USERS. Your first step is to calculate how many disk IOs it takes to do a USER_ID lookup on the current table. In the worst case, how many disk IOs will it take to use binary search to find a USER_ID in the USERS table? **Assume NO pages of USERS are in the buffer pool at the start of this query**. Please give exact IOs, and not Big-O notation. Circle your final answer.

2^{22} records / 2^6 records per page = 2^{16} pages
$\log_2(2^{16})$ = 16 IOs
2^{16} is even, so in the worst case, one more IO is required
17 IOs

e) [5 points] Having taken CS186, you know that B+ tree indexes can improve lookup performance over binary search. You want to build a B+ tree index on the USERS table with the search key on USER_ID to speed up lookups. **Assume that the root node of the tree is ALWAYS in the buffer pool** and that you use Alternative (1) for the leaves of the index, so all the full data records reside in the leaf nodes. What is the minimum fanout of the B+ tree index that would be required to support the 128 queries per second? Circle your final answer.

256 IOs per second / 128 queries per second = 2 IO per query
root is always in buffer pool, so tree can be at most 2 non-leaf levels and 1 leaf level.
$\log_{fanout}(2^{16})=2$
2^{16} pages = $fanout^2$
$(2^{16})^{1/2}$ = fanout = 2^8 = 256

Question 2 – B+ Trees (continued)

f) [5 points] Database systems often perform INDEX-ONLY scans when all the fields in the query are in the search key of the index, and scans the index data entries, without having to scan the data file. Knowing this, you create a B+ tree on the USERS table with a search key on the AGE field. You use Alternative (2) so that the data entries are only $2^4 = 16$ bytes. **Assuming all the non-leaf pages of the B+ tree are already in the buffer pool**, how many I/Os will it take to perform an INDEX-ONLY scan to read all the AGE values from USERS (say, to calculate the average age of all the members)? Circle your final answer.

2^{14} bytes per page / 2^4 bytes per record = 2^{10} record per page
2^{22} records / 2^{10} records per page = 2^{12} leaf pages
No leaf is in the buffer pool yet so the INDEX-ONLY scan requires 2^{12} IOs

g) [3 points] Given the huge performance benefits you got by doing an INDEX-ONLY scan for the "average AGE" query, the VP of Marketing suggests that you create indexes for ALL of the fields in the USERS table. Briefly state why this is probably a bad idea (i.e., what are the costs of having too many indexes?).

Having too many indexes causes too much overhead when updating/inserting/deleting from the table, since each index will have to be updated.

A less important reason is that it will use up extra disk space.

Question 3 – Hash Indexes [4 parts, 15 points total]
For parts (a) and (b) circle your answer and be sure to state why in a sentence or two. List any assumptions you are making.

a) [3 points] An extendible hashing structure can never contain overflow pages.

i) True **<u>ii) False</u>**

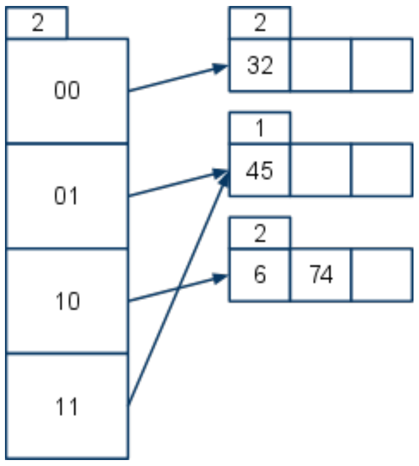
Why? *Although extendible hashing usually doesn't contain overflow pages, it is possible to have overflow pages if a large number records with the same search keys are inserted.*

b) [3 points] Your company asks you to design a hashing mechanism to index old archive data. You know that you will not performing insertion or deletions on the data, but will be querying it for equality searches on the search key. Which hashing method is most appropriate?

<u>i) Static</u> **ii)** Extendible **iii)** Linear **iv)** all are equally good

Why? *Since there is no insertion nor deletion, static hashing doesn't have the overhead of maintaining extra levels of indirection or keeping global/local depth counts. Extendible hashing may need another I/O if the table does not fit into memory. Linear Hashing needs to keep track of hash functions and next pointer. While Linear Hashing is not necessarily slower for lookups, static hashing is the most appropriate.*

c) [3 points] Consider the Extendible Hashing structure below. What is the maximum number of keys can you insert before the size of the directory must double? (no explanation needed)

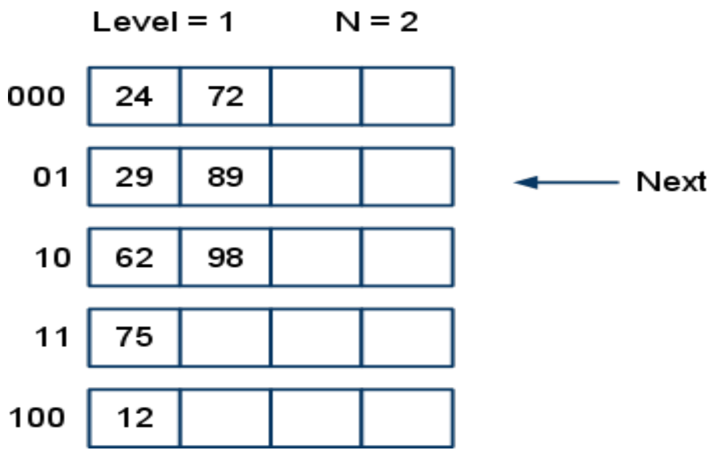


We can insert a total 8 entries.

- *2 in bucket #1*
- *1 in bucket #3*
- *2 in bucket #2*
- *Inserting another entry in #2 will cause a split, but the directory will not double. We can then add 2 more entries.*

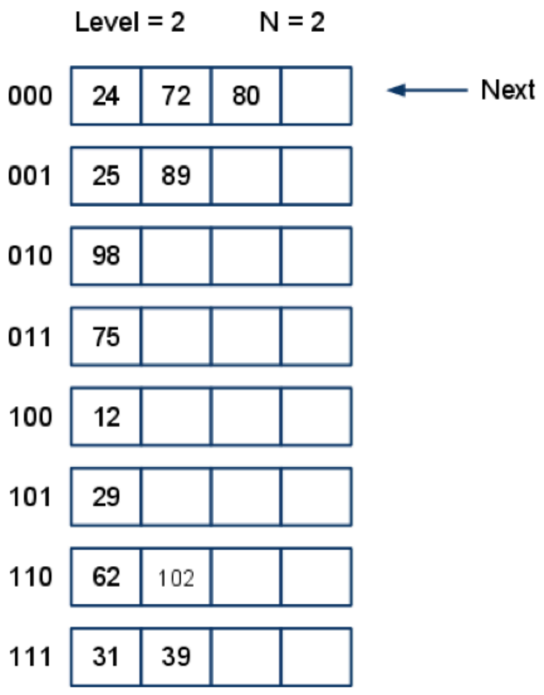
Question 3 – Hash Indexes (continued

d) [6 points] Consider Linear hashing with a hash function $h_n(x) = x \bmod n$ and the following initial state. A split is triggered when a bucket reaches a load factor of 75% (i.e., the 3rd data entry is added to a bucket). The current state of the hashing structure is depicted below.



12 =	0001100
24 =	0011000
29 =	0011101
62 =	0111110
72 =	1001000
75 =	1001011
89 =	1011001
98 =	1100010

Using the template below, show the state of the structure after inserting the following five data entries: 31 (11111), 39 (100111), 25 (11001) , 102 (1100110) , 80 (1010000). **Be sure to fill in the values of Level and N, show the Next pointer, and cross out any buckets that are not yet allocated.** (do your work elsewhere and fill in your final answer below).

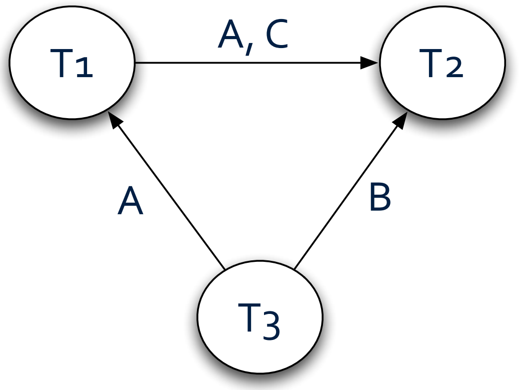


Question 4 – Concurrency Control [7 parts, 29 points total]

For parts (a-d), consider the following schedule of three transactions. Commit abbreviated "com"

Operation	1	2	3	4	5	6	7	8	9	10	11	12
T1:	R(C)		R(A)				W(A)			com		
T2:				W(C)				R(A)	W(B)		com	
T3:		R(A)			R(A)	W(B)						com

a) [6 points] Draw the dependency graph for this schedule. Be sure to list the object(s) (A, B, or C) that is (are) the cause of each dependency on each edge.


$$\begin{tikzcd} T_1 \arrow[rr, "A{,}\ C"] & & T_2 \\ & T_3 \arrow[ul, "A"] \arrow[ur, "B"'] & \end{tikzcd}$$

b) [5 points] Is this schedule conflict-serializable? If so, list a serial ordering of the transactions that would produce an equivalent schedule. If not, state why not.

Yes. T3 -> T1 -> T2.

c) [2 points] This schedule of read and write operations could be generated by a system following the regular **2PL** (two phase locking) protocol. (Circle one)

TRUE [✓ **FALSE**]

(We originally made a mistake on grading this question. We gave the points for "True", but the correct answer to this is "False".)

d) [2 points] This schedule of read and write operations could be generated by a system following the **Strict 2PL** protocol. (Circle one)

TRUE [✓ **FALSE**]

e) [4 points] In general, is Strict 2PL is more likely to encounter deadlocks than regular 2PL? State **Why** or **Why Not**.

Yes. Locks are held longer in Strict 2PL, thus increasing the likelihood of deadlocks.

f) [4 points] Is Optimistic Concurrency Control (as described in the book and in lecture) more likely to encounter deadlocks than regular 2PL? State **Why** or **Why Not**.

No. Optimistic Concurrency Control doesn't employ locks.

g) [6 points] Assuming very few people actually attended lecture, in which lock mode(s) (IS, IX, S, X, SIX) should the DBMS lock each granularity to run the following query with maximum *concurrency*?

UPDATE CS186Students SET (grade = 'A') WHERE attendedLecture=true

Granularity	**Mode (S, IS, IX, X, SIX, or *none*)**
Database	*IX*
Relation	*SIX*
Record	*X*