CSE-4411M Test #1

The Physical Database

Sur / Last Name:
Given / First Name:
Student ID:

- **Instructor:** Parke Godfrey
- **Exam Duration:** 75 minutes
- **Term:** Winter 2014

Answer the following questions to the best of your knowledge. Your answers may be brief, but be precise and be careful. The exam is closed-book and closed-notes. Calculators, etc., are fine to use. Write any assumptions you need to make along with your answers, whenever necessary.

There are five major questions. Points for each question and sub-question are as indicated. In total, the exam is out of 50 points.

If you need additional space for an answer, just indicate clearly where you are continuing.

Marking Box	
1.	/10
2.	/10
3.	/10
4.	/10
5.	/10
Total	/50

1. **The Physical Database.** *For the record.* (10 points) [Short Answer]

a. (2 points) Dr. Mark "Drop Table" Dogfurry, infamous database researcher, has the idea to keep a counter on each page of the database in order to count the number of times it has ever been fetched into the buffer pool. He is thinking this could be used as part of a new replacement policy for the buffer pool.

You think it is a bad idea, though. What would be an additional cost that Dr. Dogfurry's counter would add to the operation of the buffer pool?

This makes every page dirty*, so evey page on replacement must be flushed to disk! This is expensive.*
Also, it is unlikely such a global count would help much. The global usage has little to do with present usage, data locality*.*

b. (2 points) A disk space manager often has a method in its API

```
void allocate_page(PageId start_page_num, int runsize)
```

This lets one request a *sequence* of `runsize` number of pages to be allocated.

Why is this provided? How is it useful for system performance?

So that a physically sequential block of pages on disk can be pre-allocated. This enables sequential writes*, followed later by* sequential reads*.*

c. (2 points) If an operation is *I/O bound*, what does this mean?
What is one technique for addressing I/O-boundedness?

The process stalls periodically waiting on an I/O read or write to finish before it can proceed.
Double buffering.

d. (2 points) What is the principle of *data locality*?
Which buffer-pool replacement strategy is favoured by *data locality*?

> *That data needed for a given operation is likely to be requested again in a short time than random data from the database.*
> *LRU benefits from data locality: it replaces* least recently used *pages, leaving the* most recently used *pages, which by* data locality *are more likely to be used again.*

e. (2 points)
When might it make sense to have an index with a search key that is a superset of the table's primary (logical) key?
For instance, when would an index with the compound search key $\mathsf{A}+\mathsf{B}+\mathsf{C}$ for table $\mathbf{T}$ with primary key $\mathsf{A}+\mathsf{B}$ *ever* make sense?

> *This enables index-only accesses for queries that request A, B, and C. If such queries are common, this could be beneficial.*

2. **Index Use.** *Consumer Price Index.* (10 points) [Exercise]

Table **R** has a *clustered* tree index of type alternative #2 on A, B, C, D. (Assume **R** has additional attributes; e.g., E, F, . . .) The index pages contain 133 *index records* (encompassing 134 pointers), on average;[1] data-entry pages contain 50 data entries each, on average; and data-record pages contain 20 data records each, on average.

A's values range over 1..10,000; B's over 1..1,000; C's over 1..100; and D's over 1..10.

For Questions 2c & 2d, assume "smart" processing; that is, that the processor would minimize I/O usage with the selection information.

---

a. (3 points) An index record contains effectively the information A, B, C, D, and a pointer (an address to another page). A data entry contains effectively A, B, C, D, and an RID (which is an address to another page *and* a slot number). A data entry is *slightly* larger than an index record—by a slot number—but only slightly. So explain how it is possible there are 133 index records per page, on average, but *only* 50 data entries per page, on average.

Key compression.

---

b. (2 points) Say that table **R** has 1,000,000 records. How deep is the index tree?

Fan-out is 134. We must index 20K pages. $134^2 < 20K$ *but* $134^3 > 20K$*. So we need three levels of index pages (the root plus two more) to have page-ID "pointers" to the 20K data-entry pages.*
These three index-page layers plus the data-entry layer means the tree is four pages deep.

[1] This accounts for the *fill factor*. A page could hold more than 133 index records.

c. (3 points) Estimate the I/O cost of

select * from R where A = 1111 and B > 700;

using the index as the access path.

$1M/10K = 100$ matching A. 30% ($(1000-700)/1000$) of these match B. So, 30 matching records, in all.
Four I/O's to get to the first $A = 1111 and B > 700$ data entry. All 30 matches likely on the same data-entry page. 2 I/O's to fetch the data-record pages containing the 30 records. (Clustered, 20 records per page.) So 6 I/O's predicted.

d. (2 points) Estimate the I/O cost of

select * from R where A > 9500 and C > 90;

using the index as the access path. (Assume "smart" processing.)

$1M \cdot 500/10K = 50K$ matching records for $A > 9500$. Cannot match for $C > 90$ by the index (since B intervenes in the search key and there is no equality predicate on B in the query). But $10/100 = 10\%$ of the 50K will match also for $C > 90$ and need fetching; so 5K record fetches.
4 I/O's to the data-entry page with the first $A > 9500$ entry. Scan $20K/20 = 1K$ of data-entry pages. When $C > 90$, also fetch the record: 5K fetches costing 2.5K I/O's as we can estimate there are two matching records per fetched data-record page and the index is clustered.
So 3,504 I/O's are estimated in all.

3. **General.** *Dealer's choice.* (10 points) [Multiple Choice]

Choose *one* best answer for each of the following. Each is worth one point. There is no negative penalty for a wrong answer.

---

a. Which of the following is *false*?

A. The technology trend is that the ratio of CPU to disk I/O speed is growing over time.
B. Page size is determined by the query.
C. Sequential reads and writes are important to a database system's performance.
D. I/O time usually dominates CPU time in database operations.
E. Many records fit on a page, on average.

---

b. Physical *database independence* has the consequence that

A. applications need not know the schema of the database to compose queries.
B. applications cannot access records directly, but only via queries.
C. records from the same table have to be stored within the same file.
D. indexes must be used to access the data.
E. pointers cannot be used internally in the database system.

---

c. Relational database management systems (RDBMSs) typically implement their own buffer pool managers rather than using the operating system's (OS's) facilities *because*

A. OS's do not handle paging between disk and main memory.
B. they need control over when a page is written back to disk.
C. an RDBMS's buffer pool manager can page faster than the OS's facilities can.
D. paging by the OS has no replacement policy.
E. it is easy enough to implement, so why not?

---

d. All the following are design assumptions made in the design of relational database systems except which of the following?

A. Tables and indexes will continue to grow in size.
B. Tables may have as many columns as rows.
C. CPU operations are fast compared with I/O.
D. The physical database resides on non-volatile disk.
E. Multiple transactions can run concurrently.

---

e. Supporting variable-length records has the consequence that

A. B+ tree indexes are not possible for these records because the order of the B+ tree cannot be determined.
B. slot numbers cannot be determined as fixed addresses on the page.
C. the buffer-pool manager has to support variable length frames.
D. records from the same table may have different numbers of fields.
E. fields of the same record may have to be kept on different pages.

---

f. How many distinct search keys exist for table **T** with five attributes A, B, C, D, and E?

A. 1
B. 5
C. 31
D. 120
E. 325
F. 512

---

g. For a linear hash index, which of the following is *false*?

A. Overflow pages are needed.
B. A directory is not needed.
C. Buckets are split round robin; this has the amortized effect that overflows never get longer than one, on average.
D. The directory may double when a bucket is split.
E. The bucket that is split may not be the one that just had something added.

---

h. Consider

I. clustered tree indexes
II. unclustered tree indexes
III. clustered hash indexes
IV. unclustered hash indexes

Equality match queries can benefit from

A. Just **II**.
B. Just **I** & **II**.
C. Just **I**, **II**, & **III**.
D. Just **I** & **III**.
E. Potentially any of **I**, **II**, **III**, & **IV**.

---

i. The *external sort operator* is useful for the database system because it

A. uses no CPU.
B. is significantly faster than in-memory sort algorithms.
C. can sort collections that cannot fit in main memory.
D. can sort variable length records, while in-memory sort algorithms cannot.
E. has better "$\mathcal{O}$" ("big-Oh") than any comparison-based, in-memory sort algorithm.

---

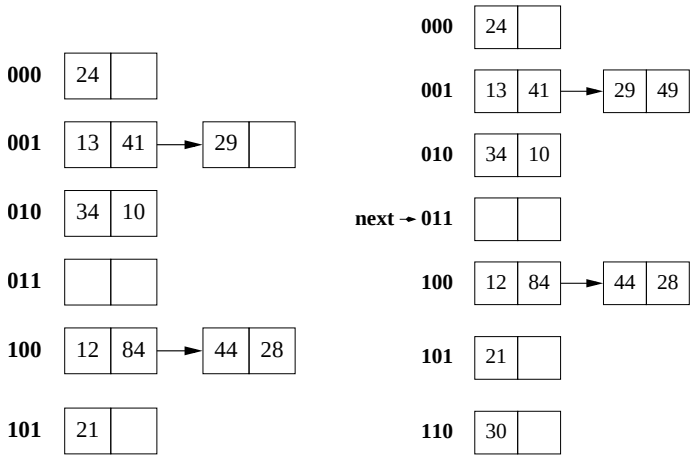
j. Using *replacement sort* instead of *quicksort* for *pass zero* of the external sort algorithm has the advantage that

A. it is faster than quicksort.
B. it allows for sequential reads, whereas quicksort does not.
C. it produces runs twice as long, on average, as the use of quicksort does.
D. it may reduce the number of I/O's for pass zero, compared with using quicksort.
E. it may reduce the number of I/O's for subsequent passes, compared with using quicksort.

---

4. **Index Mechanics.** *Go climb a tree.* (10 points) [Exercise]

Consider the following linear hash index.



(Only the hash values—and not the data entries themselves—are shown in the buckets.)

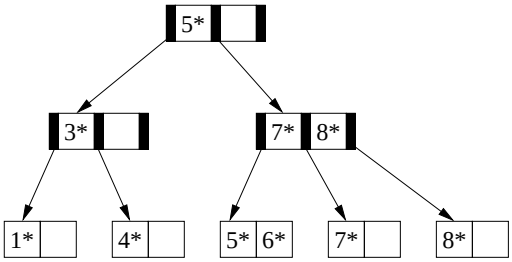
a. (3 points) Show where *osprey* (49) and *coyote* (30) would be placed in the hash index. Note that the number after each data item is its *hash value* by the index's hash function. You may show these directly in the figure above.

$49 = 110001_2$ and $30 = 011110_2$. 49 then gets added to the overflow page of 001 which has space for it.
30 gets added to 010, creating an overflow page. The creation of the overflow page causes a split. As it happens, next is pointing at 010, so it is what is split. On redistribution, 30 is what is moved to 110; 34 and 10 stay on 010.

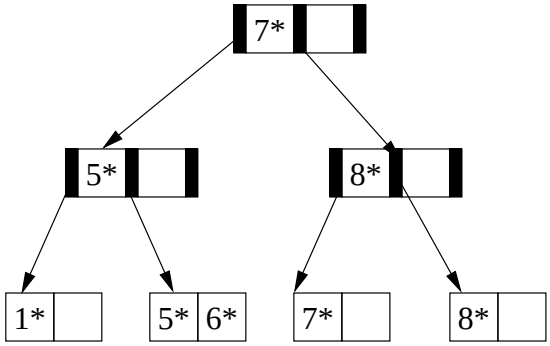
b. (2 points) Which hash-index cell is the next one that will be split? Why?

010 before 30 was added and the split was triggered; 011 after.
This is because 010 is the first that does not have a "pair" yet with the least-significant three bits.

c. (3 points) Consider this B+ tree:



Show how the tree will look if the record 4* is deleted.



d. (2 points) Consider an extendible hash of *global depth* 5. Thus, there are 32 directory slots. What is the *smallest* number of buckets that the hash index might have? (You may assume that the hash index always grew, never shrank.)
Explain.

6. Split the "same" over and over: 2 buckets of depth 5, and one each of depths 4 (2 buckets), 3 (4 buckets), 2 (8 buckets), and 1 (16 buckets), for 32 buckets in all.

5. **External Sorting.** *Hanging out with the wrong sort.* (10 points) [Analysis]

Dr. Datta Bas has developed an "improved" version of the standard external sort routine. External sorting is usually quite efficient as few passes over the records are required. However, when the buffer pool is small and/or the file to sort is huge, a more significant number of passes may be needed, requiring a read and write of every page every pass.

Dr. Bas has made the observation that we might improve performance as follows: we do not need to keep *all* the fields of the records during the sort, but just the fields that are part of the sort key *and* the *rid*!

For Dr. Bas's external sort routine, pass 0 (the initial sorting pass) and the final (merge) pass are modified. Pass 0 is modified to "project" the records, removing the unneeded fields. The final (merge) pass—call it pass f—is modified to "join" back the removed fields. Merge passes $1 \ldots f-1$ proceed just as in the regular external sort routine. The only difference is that the number of pages in the "file" for these runs is much smaller because we have removed the fields unneeded for the sorting.

Assume the buffer pool allocation is B. Pass 0 uses one buffer frame as *input* to read sequentially each page of the file to sort. It projects each record in the input frame to a record with just *search key* + *page#* + *slot#* and places it sequentially in the array of the remaining $B-1$ frames. The *page#* + *slot#* here represent the record's *rid* in the original file. When the $B-1$ frames allocated for sorting become full, the projected records are then sorted, and written out as a $B-1$ page sorted run.

Pass f must have only $B-2$ runs remaining to merge. One frame is reserved for *output* as before. $B-2$ frames are used as input to merge the (potentially) $B-2$ runs. The last frame is reserved to be used to fetch the page with the original record (from the original file) for each projected record in the merge stream, in order to retrieve the missing fields and add them back. (Recall that each projected record includes the *rid* of the original record for this purpose.)

Assume that you have file **F** to sort. File **F** is 400 pages and 5 records fit per page. The "projected" records for Dr. Bas's routine fit 50 records to a page, so after pass 0 of his routine, the "file" fits in 40 pages. Your buffer pool allocation (B) for the sort is 6 frames.

a. (3 points) First, calculate the I/O cost of sorting **F** using the basic external sort routine. (Assume that pass 0 produces runs of size B, so 6 in this case.)

Have 6 frames.

- *Pass 0: Sort blocks of six: 67 runs (66 of length 6, 1 of lenght 4)*
- *Pass 1: Merge 5 at a time: 13 runs of length 30, 1 of length 10.*
- *Pass 2: Merge 5 at a time: 2 runs of length 150, 1 of length 100.*
- *Pass 3: Merge the 2: 1 run of length 400.*

$4 \times 2 \times 400 = 3200 I/O$*'s.*

b. (4 points) Now calculate the I/O cost of sorting **F** using Dr. Bas's external sort routine. (Note that Bas's pass 0 produces runs of size $B - 1$, so 5 in this case.)

- *Pass 0′: 400 in, 40 projected out. So 8 runs of length 5.*
- *Pass 1: Merge 5 at a time: 1 run of 25 and 1 run of 15.*
- *Pass 2′: Merge 4 at a time, with fetching of records to add back fields: 1 run of 400 (full records!), plus the 2000 record fetches (at an I/O each).*

$440 + 80 + 440 + 2000 = 2980 I/O$*'s.*

c. (3 points) Under what conditions, if any, is Dr. Bas's sort routine advantageous? If none, briefly explain why not.

Few records pad each page. Savings in merge passes must outweigh the cost of record fetches in the last pass. With many records per page, this would not likely be a favourable tradeoff.

[Scratch space]

[Scratch space]

[Scratch space]

Relax. Turn in your test.