EECS-4411M: Test #1

"The Physical Database"

Electrical Engineering & Computer Science

Lassonde School of Engineering

York University

Family Name:	
Given Name:	
Student#:	
EECS Account:	

Instructor:	Parke Godfrey
Exam Duration:	75 minutes
Term:	Winter 2017

Instructions

- **rules**
  - The test is closed-note, closed-book. Use of a calculator is permitted.
- **answers**
  - Should you feel a question needs an assumption to be able to answer it, write the assumptions you need along with your answer.
  - If you need more room to write an answer, indicate where you are continuing the answer.
  - For multiple choice questions, choose *one* best answer for each of the following. There is no negative penalty for a wrong answer.
- **notation & assumptions**
  - For questions about indexes, assume that the indexes are *dense*.
  - For physical storage of records, assume row store.
- **points**
  - The number of points a given question is worth is marked. (It is worth one point, if not marked.)
  - There are five major parts worth 10 points each, for 50 points in total.

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

1. [10pt] **Hardware, I/O, Pages, & Layout.** *But I'm a software person!* SHORT ANSWER

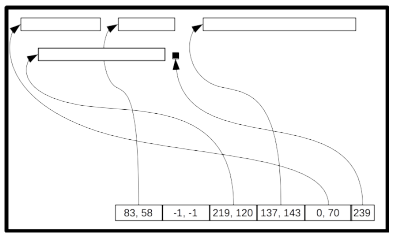
a. [2pt] You are a database administrator (DBA). You recently set up a new database system on a server with RAID. When you ran some performance measures, you find that random-access page *writes* to disk are significantly more expensive than random-access page *reads* from disk.

Provide a rational hypothesis why this might be the case.

Maybe you set the RAID up with mirroring, like 1-0. Not the best for a DB system, perhaps, but still. A read is issued to both disks in a pair that store copies of that page. The first to respond is enough. For writes, though, it has to complete to both.
Writes require writing parity too.

b. [3pt] We generally do not want to change the *slot#* of a record. This can make managing record placement on *pages* more difficult if we handle *variable-length* records.

Describe / draw a page layout strategy for records that can accommodate insertions and deletions of records on the page without changing existing records' slot#'s.



Maintain a slot array *at, say, the end of the page. Its first cell is a "pointer", an offset, to where* free space *starts on the page. Subsequent cells are pairs: the offset to that slot's record, and the length of the record. E.g., slot #0 points to a record at offset 0 and is 70 bytes long. Slot #1 points to a record at offset 137 and is 143 bytes long. Slot #3 is presently not "filled".*
This indirection allows for records to be moved on the page, but keep the same slot #.

+1pt understandable picture
+1pt slot array
+1pt records can move

SQL must support NULL values. Thus, so must the database system.

Assume that you only have to support *fixed-format* records.

---

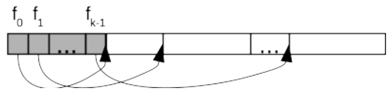
c. [2pt] If a database will have lots of NULL values, why would having support for *variable-length* records in the system be preferable to having only support for *fixed-length* records?

How ever many bytes are allocated to a field are just wasted space in a record with a NULL for that field under a fixed-length format.
In a variable-length format, a NULL can be accommodated with zero bytes used. If we are using an offset array at the beginning of each record, an offset of '0' can be used to indicate NULL, for example.

+2pt NULLs can take effectively zero space in variable-length format; take fixed #bytes in a fixed-length format, regardless.

---

d. [3pt] Briefly describe a way to represent variable-length records that may contain NULLs.



Int array at beginning of record of length k, where we've k fields. (If we have used a page layout as in Question 1b, the slot array tells us the end of the record.) f_i is the offset to the beginning of field i's bytes.
Place zero in f_i as an offset, if its "value" is NULL. First f_j with $j > i$ that is non-zero, or the record's end, indicates the end of field i.

+2pt field offset array
+1pt zeroes for NULLs' offsets, or the like.

2. [10pt] **Buffer Pool.** *Learning to swim.* Exercise

---

a. [2pt] LRU (*least recently used*) is known to be generally a good buffer-pool replacement strategy in support of most SQL operations.
Why?

It is the principle of data locality*. If a piece of data is needed for an operation, it is more likely to be needed again soon by the operation. This is generally true for SQL queries.*

2pt data locality
1pt partial explanation in right direction

---

b. [3pt] Spell out the *steps* that the buffer-pool manager needs to make to handle a *pin* call; e.g., pin(1729). Assume the replacement policy is LRU.

```
IF 1729 is in the buffer pool THEN
    remove from replacement queue (or the like)
    increment the frame descriptor's pincount
    return frame address to caller
ELSE
    find a frame to replace by replacement policy
    IF none, throw exception
    IF page in chosen frame is 'dirty' THEN write it to disk
    READ 1729 into selected frame
    set frame descriptor's pincount to 1
    return frame address to caller
```

+1pt replacement
+1pt pins, writes old page, if needed
+1pt returns frame address

c. [3pt] A *design choice* for the physical database is *how* we will *address* records to locate them. This identifier is called the *record ID* (RID). Our choice is for RID's to be *physical* or to be *indirect.*

What is the difference between these?

Physical *is when the RID is the* physical *address of the record; often, this is done as* physical page# *on disk &* slot# *on page.*
Indirect *is when the RID is assigned a unique integer. In this case, we maintian an* RID map *that stores as key-value pairs the RID with the* physical page# *&* slot#*.*
Indirect incurs the expense of maintaining and using th map, but moving records is much less expensive.

3pt good description of both
- ∗ page# & slot#
- ∗ indirect needs map

2pt weak description of one or the other
1pt vague or incomplete

---

d. [2pt] When building a database system from scratch, instead of having to implement a buffer-pool manager for our system, it would be tempting to use library calls to the operating system (OS) instead. The OS would handle paging in and out our database pages on disk to and from main memory. (Thus, this would be leveraging the OS's *virtual memory.*)

Is this a good idea? Briefly, why or why not?

Typically, it is a bad idea. We no longer control the replacement policy. What the OS does might not be good for our performance. Pinning would not guarantee the page will actually stay in main memory under virtual memory; OS may page it out. And we cannot guarantee "flush" (write to disk) when we need it.

+1pt that it is a bad idea
+1pt provides one or more valid reasons as to why

3. [10pt] **General.** *Jan, ken, pon!* MULTIPLE CHOICE

---

a. [1pt] B+ tree indexes reduce I/O cost by
  A. providing binary search.
  B. storing data records with the search keys.
  C. use of overflow pages.
  D. keeping the data records sorted.
  $\boxed{\textbf{E.}}$ increasing fan-out of the index pages.

---

b. [1pt] 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 hardware.
  C. Sequential reads and writes are important to a database system's performance.
  $\boxed{\textbf{D.}}$ CPU time usually dominates I/O time in database operations.
  E. Many records fit on a page, on average.

---

c. [1pt] How many distinct search keys exist for table **T** with six attributes A, B, C, D, E, and F? (Assume *ascending* in all cases.)
  A. 1
  B. 6
  C. 21
  D. 63
  E. 720
  $\boxed{\textbf{F.}}$ 1956

---

d. [1pt] An operation is considered to be *I/O bound* if it
  A. runs as efficiently as possible.
  $\boxed{\textbf{B.}}$ stalls frequently waiting on I/O calls to complete.
  C. uses no I/O resources.
  D. uses no CPU resources.
  E. runs entirely on disk, not in main memory.

---

e. [1pt] In database systems, hash indexes have the *advantage* over tree indexes in that
  A. they can be used for all queries that tree indexes can be, but not vice versa.
  B. they use no disk space.
  C. they can be used for *unique* search keys; that is, a search key that is also a (logical) candidate key.
  $\boxed{\textbf{D.}}$ they typically use less I/O for equality searches.
  E. they can accommodate composite search keys, while tree indexes cannot.

---

f. [1pt] Assume a B+ tree in which up to four index records (with five pointers) *or* four data records fit per page. Assume the index is primary / direct; the data records themselves are stored in the index. And assume the tree is storing millions of records.

In *worst* case, a B+ tree index may take up (approximately) how much more disk space than the data it stores?

A. $1\frac{1}{2}$ times.
B. 2 times.
$\boxed{\textbf{C.}}$ 3 times.
D. 16 times.
E. *Indefinitely more space.*

---

g. [1pt] *Double buffering* is a technique that

A. doubles *virtually* the space in the buffer pool.
B. involves reading *twice* each page fetched into the buffer pool.
$\boxed{\textbf{C.}}$ addresses I/O boundedness.
D. addresses CPU boundedness.
E. speeds up significantly individual I/O operations.

---

h. [1pt] People often use the phrase "Moore's Law" to refer to the periodic doubling of a hardware attribute.

All the following "Moore's Laws" *but* which have we observed in the industry over time?

A. The number of transistors on a integrated circuit.
B. CPU clock speed (until recently).
C. Network bandwidth speed.
$\boxed{\textbf{D.}}$ Speed of disk I/O operations.
E. Number of bytes on commodity hard disks.

---

i. [1pt] Currently, SSDs compared with HDDs (*hard disk drives*) are

I. faster
II. more reliable
III. less expensive per byte

A. *None of those.*
$\boxed{\textbf{B.}}$ Just **I**.
C. Just **I** and **II**.
D. Just **I** and **III**.
E. *All of those.*

---

j. [1pt] For database systems, RAID is used to

A. speed up transaction management, allowing more operations to run concurrently.
B. store only data for which we can tolerate loss such as indexes, but not data for which we cannot tolerate loss, such as tables.
C. provide variable-sized physical pages.
D. back up the system's databases reliably.
$\boxed{\textbf{E.}}$ survive better data loss due to hard-disk failure.

---

4. [10pt] **Index Mechanics.** *It's the carburetor, of course.* EXERCISE

Assume the search-key rule to go *left* if '<' and to go *right* if '≥'.

---

a. [2pt] You conduct some disk forensics and you discover a B+ tree structure with the data as in Figure 1.

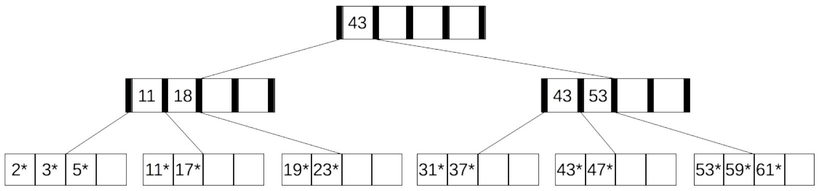


Figure 1: Recovered forensics B+ tree.

You suspect there is something wrong with this B+ tree. What is wrong?

Search keys in index pages should be moved up*, not* copied up*. So that* 43 *appears twice has to be a mistake. Indeed, it is; Note that records* 31∗ *and* 37∗ *should be to the* left *of the root, but they are not. They could not be located.*

+2pt That 43 appears twice. Or that records 31∗ and 37∗ are less than 43 but are misplaced.

---

b. [3pt] Add a record with search-key value F to the B+ tree in Figure 2.

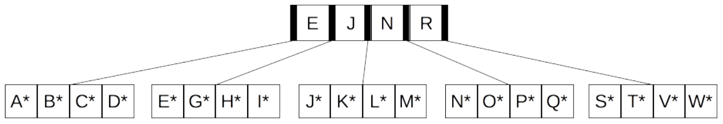
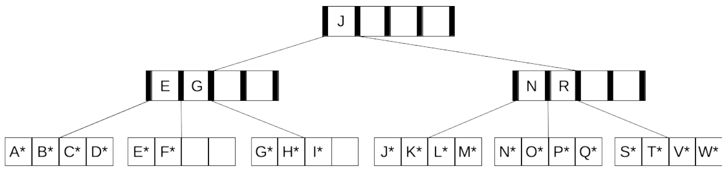


Figure 2: B+ tree for addition.



There can be other legal configurations that work. But now the tree is

- two *deep in index levels,*
- *E, F, G, H, and I were distributed, and*
- *we had a proper* copy up *of G and* push up *of J.*

+1pt each for the points above.

c. [3pt] Consider a dense B+ tree index for a table **Foo** with search key on column bar. (**Foo** has other columns too.) The search key is *not* unique. **Foo** has 10,000,000 records. The average fan-out of the tree is 200. It is secondary / indirect, so its leaf pages contain *data-entry* records. Each data-entry record contains the search-key attribute's value and an RID that provides the address to the actual record. The index is *clustered*.

50 data-entry records are on a leaf page, on average. In the file that contains the actual data records of the file, 15 records are on a page, on average.

Say that you use the index to fetch the records (with *all* their fields) from **Foo** where bar = 'Jabberwocky'. Assume that 75 records match.

Calculate the number of I/O's the fetch via the index costs. Explain the steps.

$10M/50 = 200K$ *DE (data entry) pages in total.*
So, 3 deep IP (index pages) to index those 200K DE with a fan-out of 200.
The 75 matching recors likely span 2 DE pages (holding 50 DE's each).
The 75 matching records on $75/15$ *DR (data record) pages, so 5 or 6 I/O's for that. (More likely, 6 I/O's, given spanning.)*
So $3 + 2 + 6 = 11$ *I/O's.*

+1pt for each: #IP, #DE, & #DR.

---

d. [2pt] Consider the index in Question 4c. However, this time, consider that the index is *unclustered*.

Calculate the number of I/O's the fetch costs now for the 75 records where bar = 'Jabberwocky'.

(You can just explain what are the additional or reduced I/O costs in comparison to your answer to Question 4c.)

Now an I/O to fetch each of the 75 matching records. So $3IP + 2DE + 75DR = 80$ *I/O's.*

2pt 80 I/O's
1pt something reasonable in explanation, even though wrong answer.

5. [10pt] **Design Choices.** *Wow! Much choice!* Analysis

The Toronto-based start-up company *Flash Bash* has just announced that they have created a new flash *main memory.* Their marketing claims the following.

- Faster (slightly) than current main-memory technologies!
- Random access!
  (Per byte, *not* per page, just like other main-memory technologies.)
- Less expensive than disk, per byte!
- Non-volatile (like disk)!

They claim their memory technology is robust. (That is, it will last a long time without faults.) They plan to start producing 1-, 2-, and 5-terabyte chips next month, and the price will be comparable to disk, per byte.[1]

Infamous database researcher Dr. Mark Dogfurry has incorporated *Flash-IT* to build a new SQL-based relational database system that takes advantage of *Flash Bash*'s new memory technology. He has hired you (with stock options!) on the team to build this new database system.

The system will run on a new server machine *without* traditional main memory and disks; instead, it will have a single, large Flash-Bash memory.

---

a. [2pt] Do we still need a buffer pool and a buffer pool manager?
Briefly, why or why not?

No, a buffer pool is a memory cache *that caches pages from secondary storage (e.g., disk) into primary memory. Now, everything is in primary memory / storage already.*

+1pt "no"
+1pt BP is a memory cache; nothing to cache now

---

b. [3pt] Would we still need to support tree indexes?
If so, should we implement B+ tree indexes?
Briefly explain.

Yes, we still need indexes. They significantly speed up search by value. And trees let us do range searches, which we need to support. We cannot always just do binary search, because we may want more than one index per table. If we do not change the design rule that records are stored just once, only one can be "clustered". We would not *need B+ trees; these optimize around expensive I/O, which we no longer have. Any binary search tree—like, say, red-black trees—would suffice.*

+1pt yes, still need indexes; speed up search
+1pt yes, need tree indexes to support range queries
+1pt no, do not need B+ trees

[1] I am making all this up, unfortunately.

c. [2pt] Would really long records still be a design concern for us?
Briefly, why or why not?

Not really. Things now do not need to be pagified. So we do not have to worry about page boundaries. We can manage all our main memory (which is also now our storage!) as if it were one giant page, including free space. Clustered vs unclustered is not longer much of an issue (except still requiring indirection, as for having multiple indexes per table); followng a pointer to the record is very inexpensive.

+1pt "no"
+1pt things no pagified; following a pointer is inexpensive

---

d. [3pt] Would we still need *record IDs* (RIDs)?
Briefly explain.

Yes. We will still want to support multiple indexes per file / table. We likely want to keep our design principle that a record is stored just once, if only to economize on space usage. Therefore, we have to support indirect indexes with "data entries" that point to the records. This is an RID. Note that an RID is just a memory address now, under this scenario. (Note that we might want to go with indirect RIDs now, as in Question 2c, along with an RID map; this would lower expense when a record has to be moved.)

+1pt "yes"
+1pt need multiple indexes
+1pt RID is a memory pointer now.

Extra Space

Did you see your shadow?

Extra Space

Happy Groundhog Day!

Extra Space

You reached the end. Turn in your test. Return to the wild.