# CSE-4411(A) Test #1

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

- **Instructor:** Parke Godfrey
- **Exam Duration:** 75 minutes
- **Term:** Fall 2005

Answer the following questions to the best of your knowledge. Be precise and be careful. The exam is closed-book and closed-notes. 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.

**Regrade Policy**

- You should not write anything on the exam paper after the exam period, if it is to be considered for regrading. Write any subsequent notes on separate paper.
- Regrading should only be requested in writing. Write what you would like to be reconsidered. Note, however, that an exam accepted for regrading will be reviewed and regraded in entirety (all questions).

| Grading Box | |
|---|---|
| **1.** | /10 |
| **2.** | /10 |
| **3.** | /10 |
| **4.** | /10 |
| **5.** | /10 |
| **Total** | /50 |

1. (10 points) **Buffer Pool Manager.** *How popular do you want to be?* [analysis]

   Dr. Mark Dogfurry, the infamous database designer at *Fanatics-R-Us, Inc.*, has come up with a new idea to improve the performance of the buffer manager in database systems. He suggests that space for two *long integers* (say 64 bits each) be reserved on each data page. The first will be for a field called *hits*, and the second for a field called *creation*.

   When the page is created, its *creation* field is set to the current date (say measured in seconds from midnight 1 January 1970 as is done in UNIX). Thus, *creation* is a time-stamp for the page. The *hits* field is initially set to zero.

   Every time the page is pinned, its *hits* field is incremented by one. (When the page is unpinned, however, the page's *hits* field is *not* likewise decremented.) Thus the *hits* field counts how many times the page has been pinned over its lifetime.

   Dr. Dogfurry claims a page's *popularity* can be measured as

$$pop = hits/(now - creation)$$

   where *now* is the time-stamp for the present moment.

---

   a. (3 point)

   b. (3 points) Design a buffer manager replacement policy that beneficially employs pages's *popularity*. Why would this likely be a good policy?

*We can keep the unpinned pages—that is, pages with pincount == 0—in a priority queue based on the pages's pop values. We insert a reference to the frame containing a recently unpinned (to pincount == 0) page by its pop value at* that *time. Whenever we pop the priority queue for a victim frame, its pop is the smallest of those in the queue.*
*Note that this strategy is similar to LRU. Instead of using the time when the page was unpinned (to 0), though, we use pop. This also requires a priority queue instead of a queue, since a recently unpinned page's pop value may be smaller than some already queued. Note that it takes $\mathcal{O}(\log Q)$ to insert (given the priority queue is reasonably implemented) and popping is also $\mathcal{O}(\log Q)$ as the priority queue has to be adjusted. (Here, Q is the length of the queue.)*
*Some folks noticed a page's pop is always changing, since now is involved. This could make the "queue" management quite expensive if we needed to continually check and adjust the pages's ordering. We compromised above by using the page's pop* when *it is inserted, and not adjusting it while it is in the queue.*
*Dr. Dogfurry's pop with our strategy above could likely be good as it favours keeping pages that are often pinned in the buffer pool over pages that are not often pinned. Indeed, this is the objective of a buffer-pool replacement policy, to keep pages around that are likely to be pinned again soon, thus reducing overall I/O expenditure.*

c. (4 points) List several disadvantages to Dr. Dogfurry's *popularity* scheme.

*Major disadvantages:*

- *Since we store some bookkeeping data (pop) on the page, and must update that data every time the page visits the buffer pool (BP), every page in the BP is always* dirty. *This is major overhead as we must write each page when its frame is reused.*
- *Popularity is not the same as* locality. *A page that might have been needed frequently in the past may no longer be, and vice-versa. However, we will favour pages that used to be popular for a long time. A good strategy should be based on locality instead.*

*Other disadvantages:*

- *We waste some valuable space on each page for our bookkeeping that, otherwise, could be used for data.*
- *The replacement strategy is more complex and expensive than others (e.g., LRU). Since this is a core operation, we need it to be fast.*

d. (3 points) Present *in brief* a modification of Dr. Dogfurry's popularity scheme that might make for a good buffer pool replacement strategy based on popularity, but would not suffer from the problems you identified in your answer to Question 1c.

*Do not keep the popularity counter on the page, but in the frame descriptor of the frame holding the page while it is in the BP. The counter information is lost for the page when it is removed from the BP; however, a page is never made dirty due to the counter.*

*This would probably be better than the original strategy in Question 1b because it is closer to the idea of locality since a page's popularity is measured with respect to the current duration of its stay in the BP, rather than over its entire life.*

I was just looking for a sensible answer here with respect to your answers to Questions 1b & 1c. So an alternate answer might suffice here.

2. (10 points) **General.** *Rock, scissors, paper.* [multiple choice]

For each of the following, choose *one* best answer.

a. (1 point) Consider an extendible hash with $2^{10}$ directory slots. It has *at most* how many buckets?

**A.** 1
**B.** 10
**C.** 11
**D.** $2^{10} - 2$
[✓ **E.**] $2^{10}$

b. (1 point) Consider an extendible hash with $2^{10}$ directory slots. It has *at least* how many buckets?

**A.** 1
**B.** 2
**C.** 10
[✓ **D.**] 11
**E.** $2^{10}$

c. (1 point) Why are overflow chains in a linear hash of length one, on average?

**A.** The question is nonsense. Overflow pages are not needed in a linear hash.
**B.** A bucket is split when it overflows by doubling the directory.
[✓ **C.**] Buckets are split round robin; this has the amortized effect that overflows never get longer than one, on average.
**D.** If a record is hashed to a bucket that is overflowed, the record is rehashed.
**E.** Overflows are redistributed immediately.

d. (1 point) How many distinct search keys exist for table **T** with the five attributes A, B, C, D, and E?

**A.** 1
**B.** 5
**C.** 32
**D.** 120
[✓ **E.**] 325
**F.** 3125

e. (1 point) Variable length fields mean records are variable length. This has the consequence that

[✓ **A.**] slot#'s cannot be determined as fixed addresses on the page, so a slot directory on each page is necessary.
**B.** the buffer pool manager must support variable length frames.
**C.** different records from the same table can have different numbers of fields.
**D.** the different fields of the same record must be kept on different pages.
**E.** B+ tree indexes are not possible for these records because index entries would be variable length.

---

f. (1 point) B+ trees are more suitable than balanced binary search trees for database indexes because

**A.** it is impossible to maintain the balance of a binary tree when it is disk based rather than main-memory based.
$\boxed{\textbf{B.}}$ the tree is shallower because of higher fan-out.
**C.** all the leaves are the same distance from the root.
**D.** the tree grows by splitting up, not by adding new leaves below.
**E.** binary search trees cannot accommodate composite search keys, but B+ trees can.

---

g. (1 point) Using replacement sort instead of quicksort for pass zero of the external sort algorithm has the advantage that

**A.** it is faster than quicksort (even though they are the same $\mathcal{O}$-wise).
**B.** it allows for block reads whereas quicksort does not.
$\boxed{\textbf{C.}}$ it produces runs twice as long, on average, as the use of quicksort does.
**D.** it reduces the number of I/O's for pass zero (compared with using quicksort).
**E.** it reduces the number of I/O's for subsequent passes (compared with using quicksort).

---

h. (1 point) A relational database system implements an external sort routine because

**A.** it is significantly faster than other standard sorting routines like quicksort.
**B.** it is used by tree indexes whenever a new record is inserted.
$\boxed{\textbf{C.}}$ it is needed to sort files too large for main memory.
**D.** it can sort entirely on disk without using main memory.
**E.** standard sort routines cannot sort for compound search keys.

---

i. (1 point) Page size is determined by

**A.** the programmer.
**B.** the operating system.
**C.** the Internet.
**D.** the database system architecture.
$\boxed{\textbf{E.}}$ the hardware.

---

j. (1 point) All the following are design assumptions made in the design of relational database systems except which of the following?

**A.** Many records fit per page.
$\boxed{\textbf{B.}}$ Each table can fit in main memory.
**C.** Tables have many records but few columns (in comparison).
**D.** I/O is expensive compared with CPU operations.
**E.** Main memory is volatile.

3. (10 points) **Index Mechanics.** *To hash or not to hash.* [short answer]

a. (3 points) As you know, it is not possible to have *two* clustered indexes on the same table. This is because, by design, the database system keeps a *single* copy of the data records themselves.

You are working on a database design with your boss, Dr. Dogfurry. He says that for table **T**—with attributes A, B, C, D, and E—two clustered indexes are required: A+B+C and D + B.

What can you do to effectively have two "clustered" indexes? That is, any query that would benefit from one or the other index above, *if clustered*, would be nearly as efficient to evaluate under your "fix".

*Make D + B the clustered index. Make A + B + C + D + E an* unclustered tree *index. Any query that could have benefitted from a clustered index on A+B+C could benefit nearly as well using A + B + C + D + E via an index-only plan. And it can* always *be used as index-only since it contains* all *of* **T***'s columns. This is slightly less efficient since the index on A + B + C + D + E will be a deeper tree than for A + B + C.*
*Likewise, we could make A+B+C the clustered index, and an unclustered tree index on D + B + A + C + E.*
*In a way, we are keeping two copies of the records! A data entry of the unclustered tree index contains the contents of the entire record.*

b. (3 points) Consider a hash index based on linear hashing. The basic *split rule* is to make a new hash cell whenever an overflow bucket is generated.

Should this policy be modified when there are many duplicates? For instance, when an overflow page has occurred because of a new search-key entry, but the search-key value is identical to the search-key values already in the bucket, should a split be made? Why or why not?

*Yes, the splitting policy probably should be modified. Overflows due to duplicates are different than overflows due to key clashes. Splitting will redistribute records that are in the same bucket due to key clashes, but it will never redistribute records that have the same key values.*
*The index needs to offer efficient access to records of a given key value. Adding extra hash cells because of additional records with duplicate values does not improve access efficiency. Instead, we could end up with near-empty buckets, splitting too often, and wasted space. So a better policy would be not to split in this circumstance.*
*Caveat: If the bucket in question has a mix of keys and many entries of the given key (instead of all the entries's keys being the same), then a split might be warranted as, when this bucket's splitting turn comes around, the different keys could be redistributed. This would yield improved access to those entries with other key values.*

Whenever we create a primary key constraint, e.g.,

```
alter table T add primary key (a, b);
```

the database system creates automatically an index on the key (e.g., A + B).

c. (2 points) Why does the system do this?

*The system does this so that it can check* entity integrity*—that is, for violations of the primary key constraint—efficiently on INSERTs and UPDATEs to the table. It also needs this for checking* referential integrity*—that is, for violations of foreign key constraints—efficiently.*
Many answered that it is because this index would likely be useful for many queries. This is a valid observation, and quite likely true. However, it does not explain why the *system* itself would benefit from such an index.

d. (2 points) Assume that the table will be very large (that is, it will have many records). Should the index that the system creates be a tree index or a hash index? Briefly, why?

*It should be a hash index. The system need the index to check for primary and foreign key violations. These require equality probes on the table. A hash index will be more efficient than a tree index for equality probes.*
Many said a tree index since it could be used for both equality and range queries. However, when does one ever naturally ask a range query on the key?
There is a reason why we might want this to be a tree index, but it is not to support range queries. We hadn't come to this reason yet in the course, and it wouldn't appropriately answer this question.

4. (10 points) **Index Usage & I/O's.** *May I borrow an I/O or two?* [exercise]

```
select st#, name
    from Student
    where age > 24 and major = 'CompSci'
```

There are 50,000 records in **Student**. Its primary key is st#. The file that stores the records of **Student** has 20 records per page, on average.

Assume that the predicate "major = 'CompSci'" alone matches 2,000 records, the predicate "age > 24" alone matches 20,000 records, and together—as a conjunction—they match 800 records.

The following are the indexes available for the table **Student**. Each is a tree index of type alternative #2.

**A.** st#
- clustered
- 100 data entries are on a leaf page (on average)
- fan out: 120, depth: 2 (not counting the leaf layer)

**B.** major + name
- 75 data entries are on a leaf page (on average)
- fan out: 100, depth: 2 (not counting the leaf layer)

**C.** age + st# + name + major
- 50 data entries are on a leaf page (on average)
- fan out: 60, depth: 2 (not counting the leaf layer)

**D.** major + age
- 90 data entries are on a leaf page (on average)
- fan out: 120, depth: 2 (not counting the leaf layer)

For each index, state whether it can be used to evaluate the query. If so, calculate the I/O cost of using it.

Also calculate the cost of a filescan.

Which is the best solution? Index **A**, **B**, **C**, **D**, or a filescan?

(Space for the answer to Question 4.)

*Index **A** is not useful since the query's conditions do not match in any way the index's search key. A filescan would be better.*
*A file scan would cost* $50,000/20 = 2,500$ *I/O's. (Usually, an index's bookkeeping information gives us immediate access to the leftmost, and rightmost, leaf pages, so we do not have to transverse the index pages to accomplish a filescan via an alternative #1 index, which index **A** is.)*
*Index **B** matches partially the conditions, namely the predicate on major. So we can access through the index all the records with major = 'CompSci'. We can only check the unmatched predicate age > 24 once we have the records already in memory ("on the fly"). So this costs 2 index page fetches,* $\lceil 2,000/75 \rceil = 27$ *data-entry page fetches, and likely 2,000 data-record page fetches, one per record we need to retrieve. The total is 2,029 I/O's.*
*Index **C** matches partially the conditions, just on the age predicate. Yes, the index's search key does also include major; however, because of the other components in the search key (st# and name) in between, we cannot prefix-match both predicates to the search key. The search key does include* all *the columns we need from* **Student***, so we can use it as index-only. So this costs 2 index page fetches, and* $\lceil 20,000/50 \rceil = 400$ *data-entry page fetches. Since we can use this as index-only, there are not data-record fetches. The total is 402 I/O's.*
*Index **D** matches* both *predicates. So this costs 2 index page fetches,* $\lceil 800/90 \rceil = 9$ *data-entry page fetches, and 800 data-record fetches, one to retrieve each matching record. The total is 811 I/O's.*
*Index **C** came out best.*

5. (10 points) **External Sort.** *Dog sort.* [analysis]

   Assume that $B + 1$ buffer frames are available for sorting.

   Dr. Dogfurry has designed a new external sort algorithm:

```
Phase 0 (quick sort phase):
    while there are unprocessed pages from the input file
        read in $B + 1$
            (or the remaining number of unprocessed pages, if less than $B + 1$)
            unprocessed pages
        quicksort over the records of the read-in pages
        write out the sorted run
Phase 1 (merge phase):
    while number of unprocessed runs $> 1$
        merge the $B$ oldest unprocessed runs into a new run
```

   An *unprocessed* page is one that has not yet been sorted yet. An *unprocessed* run is one that has not been merged into a new run yet. Each run is timestamped. So the oldest run is the one that was first created.

   He is proud of the new algorithm because it sorts a file in just two *phases*, whereas the basic external sort algorithm may need to make more than two *passes*.

---

a. (5 points) Is Dr. Dogfurry's algorithm the same as, nearly the same as, or different than, the standard external sort algorithm? Explain.

*It is nearly the same. The algorithm's "phase 0" is the same as* pass *0 of the standard algorithm. Dr. Dogfurry's "phase 1" is essentially eqivalent to* all *the merge* passes *of the standard algorithm. He just doesn't distinguish the dividing line when we have finished merging all the runs generated in the previous pass and when we are starting to merge the runs of the next pass. Choosing the oldest passes to merge is essentially what the standard algorithm (as expressed in our textbook) does.*

*The slight difference is as follows. The last merge of a pass under the standard algorithm may not be "full". That is, say we are doing k-way merges: there may not be enough runs left for the last merge to be k-way. So the algorithm will merge the remainding runs, even though there are not k of them.*

*Dr. Dogfurry's algorithm will always do k-way merges, except for possibly the very last merge. At the end of a "pass", if there are not k runs made in the previous "pass", the algorithm will borrow enough runs made at the beginning of this "pass" to bring it up to a k-way merge.*

*If you think about it, it might seem this is always an improvement. Indeed, some of the time, this will save I/O's. However, it does not always. In fact, there are situtations when Dr. Dogfurry's algorithm will spend* more *I/O's than the standard algorithm would.*

Think about this. This would be an excellent final exam question.

b. (5 points) The algorithm above can be said to use a *least recently made* (LRM) policy: for each merge, it chooses the $B$ *oldest* unprocessed runs to merge.

Dr. Dogfurry claims that, actually, a *most recently made* (MRM) policy would be far better. I.e.,

**merge** the $B$ *newest unprocessed* runs

Explain whether the MRM version of his algorithm is more efficient or less efficient than the standard external sort algorithm.

*This is disasterous. It is much less efficient. The standard algorithm, and Dr. Dogfurry's LRM-version algorithm, are both $\mathcal{O}(n \log n)$. The MRM-version of Dr. Dogfurry's algorithm is $\mathcal{O}(n^2)$.*
*Why? Consider $B = 2$ for now. Consider we have $N$ runs after "phase" 0, each $B$ long.*

- *Merge 1: Makes a run of size $2B$.*
- *Merge 2: Merges the newest run (length $2B$), made by merge 1, and the next newest (which is from from phase 0 and of length $B$) to make a run of length $3B$.*

  $\vdots$
- *Merge $N-1$: Merges the newest run (length $(N-1)B$), made by merge $N-2$, and the next newest (which is the last run made in phase 0, and is of length $B$) to make the final run of length $NB$.*

$$2B \sum_{i=1}^{N-1} (i+1) = 2B \sum_{i=1}^{N-2} i = B(N-1)(N-2) \approx BN^2$$

*Horrible! The damage is less severe when $B > 2$, but it remains $\mathcal{O}(N^2)$ for any $B$, and this will be significantly worse than the LRM or the standard version.*

(Scratch space.)

(Scratch space.)

(Scratch space.)

Relax. Turn in your exam. Go home.