

COSC-4411(A) Midterm #1

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

- **Instructor:** Parke Godfrey
- **Exam Duration:** 75 minutes
- **Term:** Fall 2004

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

- 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.** *Dirty laundry.* [analysis]

- a. (5 points) Dirty pages (`dirty_bit = true`) cost more than clean pages (`dirty_bit = false`) to replace because they need to be written back to the disk.

Design a replacement strategy that favours replacing clean pages.

Be careful not to favour picking clean pages as victims too much. Dirty pages should be picked occasionally, even when unpinned clean pages are available. You do not want hot clean pages always being replaced while cold dirty pages stay.

One idea: Modify the clock replacement strategy. Instead of keeping a reference bit per frame, we shall keep a reference count (`int ref_count`). When a dirty page becomes completely unpinned (`pin_count = 0`), set its `ref_count` to 2. When a clean page becomes completely unpinned, set its `ref_count` to 1.

For clock to choose a frame to use for replacement, its `ref_count` must be zero. If the current frame clock points to has (`ref_count = 0`), clock decrements `ref_count`, and goes to the next frame.

This will leave dirty pages in the buffer pool one more cycle of the clock than for clean pages, thus favouring leaving dirty pages in the buffer pool for longer.

-
- b. (2 points) Can a clean page that is unpinned (`pin_count = 0`) become dirty? How so or why not?

No. A page can become dirty only if an X-act that requested it writes on it. For that to happen, the X-act should have the page pinned.

-
- c. (3 points)

Prefetching is picking up opportunistically some extra pages off the disk that are physically contiguous after the requested page, and bringing them into the buffer pool along with the requested page. This has cost advantages since the sequence of pages can be read in one sequential read by the disk space manager, thus in one I/O. This is a win if the extra pages are soon requested since they are already in the buffer pool.

Would accommodating prefetching by the buffer pool manager be easy or be hard? Would the replacement strategy have to be modified much to handle it?

This is somewhat difficult. Say that we want to bring in five contiguous pages off of disk. We need five contiguous frames in the buffer pool to place them. Hardware-wise, that is how a sequential read works. The disk-space manager tells the disk-controller card, "Copy from this disk address this number of pages to this start address in main memory." Thus the replacer would have to identify five adjacent pages in the buffer pool eligible for replacement. This would complicate the replacement policy quite a bit.

Another approach is for the buffer pool to have a "staging area." The pages are read into the staging area, and then moved to appropriate frames. The extra main-memory copies involved would not add much overhead.

Another important issue is that the replacement strategy should be adjusted to leave prefetched pages in the buffer pool for a while. These prefetched pages have `pin_count = 0` since no one has actually requested / pinned them yet. So, otherwise, they are in danger of being immediately replaced. However, if they are replaced before anyone requests them, this defeats the whole reason for doing the prefetching.

2. (10 points) **Indexes and Cost.** *That better be your index finger...* [exercise]

```
SELECT * FROM Foo
  WHERE level = 3
     AND 20 < bar AND bar <= 30
     AND 100 < flavour AND flavour <= 250;
```

Table **Foo** contains 1,000,000 records. There are 20 records per page, on average, for 50,000 pages.

- Values of level: 1, 2, 3, 4, & 5.
- Values of bar: 1, 2, ..., 100.
- Values of flavour: 1, 2, ..., 1000.

For each value, assume there are an equal number of records with that value (uniformity assumption). So 200,000 records have level = 2, and 10,000 records have a flavour value between 120 and 130. Assume also that the values of one column are not correlated with the values of another column (independence assumption).

Indexes. All are of alternative 2). Each has a depth of 3, not including the data record / entry pages.

- I.** An unclustered tree index on bar, level.
- II.** A clustered tree index on level, bar.
- III.** An unclustered tree index on flavour.

a. (2 points) If we used no indexes, how many I/O's would it cost to evaluate the query?

We have to read in every page of the file: 50,000 I/O's.

- b. (2 points) Are indexes **I** (bar, level) and **II** (level, bar) equivalent for this query? Why or why not?

*No. We have an equality condition on level and a range condition on bar in the query. So we can use **II** to retrieve those records satisfying both conditions; but we can only use **I** to retrieve records satisfying the bar condition (so retrieving many more records), but then must check the level condition on-the-fly. And, of course, **I** is unclustered while **II** is clustered. That will be a big difference for this query.*

- c. (6 points) Determine the I/O cost for each of the three cases using the index to evaluate the query.

Which is best?

I Retrieves all records such that $20 < \text{bar} \leq 30$. This is $10\% \left(\frac{30-10}{100-0}\right)$ of the records. So 3 I/O's of index pages. Assumption: Say 50 data-entry (DE) pages per data-entry page (leaf pages of the tree index). So there are 20,000 DE pages. We read 10% of them, so 2,000 I/O's. For each data entry, we can check whether $\text{bar} = 3$ before fetching the record, since bar is part of the index's search key. So we only fetch the record if $\text{bar} = 3$. How many qualify? $20\% \left(\frac{1}{5}\right)$. So of the 100,000 data entries we have fetched, we will fetch the records for $\frac{1}{5}$ of them, so 20,000 records. Because the index is unclustered, this is 20,000 I/O's. So the grand total is 22,003 I/O's.

***II** Retrieve all records such that $\text{level} = 3$ and $20 < \text{bar} \leq 30$. This is $\frac{1}{10} \times \frac{1}{5} = \frac{1}{50}$, 2%, of the records. So 3 I/O's of index pages and 1 I/O for the DE page. Of the 50,000 data-record pages, 2% qualify, so 1,000 pages. So we need to read 1,000 data-record pages. The grand total is 1,004 I/O's.

III Retrieve all records such that $100 < \text{flavour} \leq 250$. This means $15\% \left(\frac{250-100}{1000-0}\right)$ qualify. So 3 I/O's of index pages. We would need to read 3,000 DE pages (assuming 50 data entries per page). For each qualifying record ($15\% \times 1,000,000 = 150,000$), it costs a single I/O to fetch, as the index is unclustered. The grand total is 153,003 I/O's.

Notice a filescan of the 50,000 pages would have been better here.

3. (10 points) **Miscellaneous.** *Random Access.* [multiple choice]

Indicate one best answer for each.

a. *Blocked I/O* saves what?

- A.** I/O cost by reading and writing sequences of pages arranged sequentially on disk much faster than reading and writing them one at a time.
 - B.** Space in the buffer pool because only one page of each file being used in the current operation needs to be present at any given time.
 - C.** Time by prefetching pages that might be needed into the buffer pool ahead of time.
 - D.** Disk space by packing variable-length records on a page more efficiently.
 - E.** The I/O's of writing out a temporary result table and reading it in again when those results can be immediately used by the next operation.
-

b. Variable length fields mean records are variable length. This has the consequence that

- A.** the buffer pool manager must support variable length frames.
 - B.** slot#'s cannot be determined as fixed addresses on the page, so a slot directory on each page is necessary.
 - 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 the order of the B+ tree cannot be determined.
-

c. Which of the following changes would *not* make researchers reevaluate standard database system design?

- A.** Cheap, fast, non-volatile main memory becomes standard.
 - B.** Computers are standardly configured with a terabyte of RAM.
 - C.** Databases with hundreds of columns per table become normal.
 - D.** Optical (non-volatile) secondary memory with significantly better I/O speeds than hard-disks become available.
 - E.** CPU speeds increase by a factor of ten while everything else stays the same.
-

d. Physical database independence has the consequence that

- A.** pointers (references) cannot be used internally in the database system.
 - B.** records from the same table must be stored together in the same file.
 - C.** users need not know the schema of the database in order to compose queries.
 - D.** users cannot access records directly, but only via queries.
 - E.** indexes must be used to access the data.
-

e. 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.
- B.** all the leaves are the same distance from the root.
- C.** the tree is shallower because of higher fan-out.
- 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.

-
- f. Alternative #1 for index organization is not usually implemented in practise for tree indexes because
- A. data records are bigger than data entries, so fan-out is reduced.
 - B. when records are redistributed on leaf-page splits, their rid's can change.
 - C. if the data records are part of the B+ tree, no second index on the table is possible.
 - D. it is not possible to do key compression under alternative #1.
 - E. using alternative #1, composite search keys cannot be accommodated.
-

- g. Consider
- I. unclustered tree indexes
 - II. clustered tree indexes
 - III. unclustered hash indexes
 - IV. clustered hash indexes
- Range queries can benefit from
- A. Just I.
 - B. Just I & II.
 - C. Just I, II, & IV.
 - D. Just II & IV.
 - E. Potentially any of I, II, III, & IV.
-

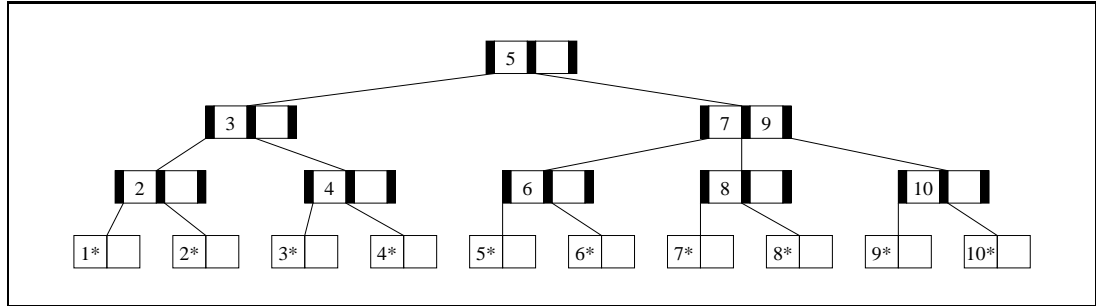
- h. The buffer manager manages for the database system
- A. disk memory.
 - B. main memory.
 - C. file allocation and deallocation.
 - D. query processing.
 - E. lock management for transaction processing.
-

- i. Which of the following is *false*?
- A. Locating a record by key in a sorted file by binary search versus locating it via a B+ tree requires many more key comparisons.
 - B. Locating a record by key in a sorted file by binary search requires more I/O's than locating it via a B+ tree, in general.
 - C. A bulk build of a B+ tree is faster than building it by inserting a record at a time.
 - D. If the data records are kept in a sorted file, it is still useful to have a B+ tree index based on the same search / sort key.
 - E. If there is an unclustered B+ tree index over the data records, this does not mean that the records are necessarily sorted.
-

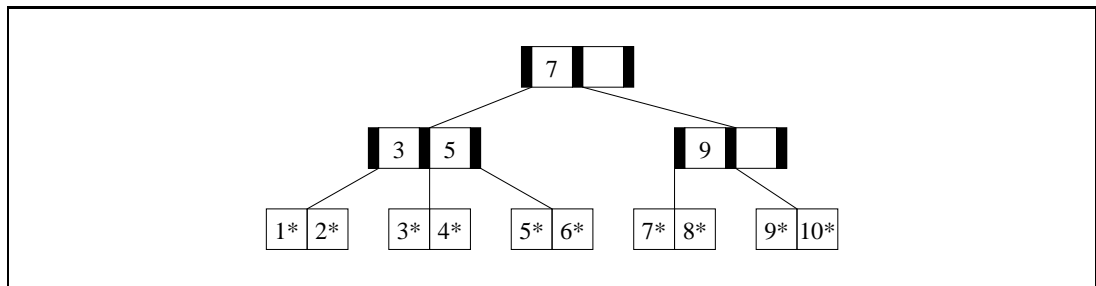
- j. Which of the following is *false*?
- A. The technology trend is that the ratio of disk I/O speeds to CPU speeds is getting smaller over time.
 - B. Page size is chosen to best suit the application.
 - C. Generally, many records fit on a page.
 - D. Sequential reads and writes are important to a database system's performance.
 - E. Generally, I/O time dominates CPU time in database operations.

4. (10 points) **B+ Trees.** *How low can you go?* [short answer / exercise]

- a. (3 points) Consider a B+ tree of order one. Construct an example of B+ tree with the *worst* (largest) possible depth that indexes 10 data records.

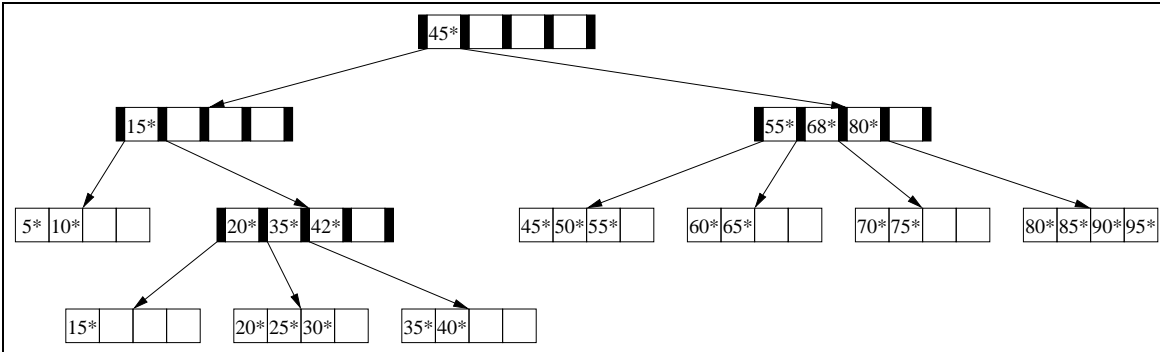


- b. (3 points) Consider a B+ tree of order one. Construct an example of B+ tree with the *best* (smallest) possible depth that indexes 10 data records.



Some people have argued that for 3a, C is also a good answer. It is true that blocked I/O is used for prefetching. However, prefetching and blocked I/O are different. Prefetching still can be advantageous, even if not done with blocked I/O. And block I/O is used for many things in databases than prefetching. A is a correct answer, and a better answer than C.

c. (4 points) Identify four distinct problems with the following “B+ tree”.



- i. Not all nodes are at least half full.*
- ii. Record 55 is out of place. By the apparent convention in the tree, it should be to the right of key 53 in the index node. (Otherwise, many other records are misplaced.)*
- iii. In a B+ tree, all leaves should be at the same depth. This is not the case here.*
- iv. There is no leaf page to the right of key 42. Either the key should not be there, or there should be a leaf page.*
- v. Keys in index pages are marked with *'s, indicating these are data records themselves. Data records are not stored in index pages in a B+ tree.*

5. (10 points) **Linear Hash Indexes.** *Coming up empty.* [analysis]

- a. (5 points) Dr. Dogfurry, infamous database researcher, has noted that sometimes when a new bucket creation is triggered—whenever an overflow page has been made—*no* keys redistribute into the new bucket. He reasons that it does not make sense to make the new bucket in such cases.

So the algorithm should check whether redistribution would put anything into the new bucket; if not, a new bucket is not made, and `next` is not advanced.

What is wrong with Dr. Dogfurry's change? What bad consequences would this have?

This is quite bad. The #buckets will not grow until a record is added to that very bucket in question that would result itself redistribute to the new bucket if this bucket were split. In the meantime, long overflows can develop on the other buckets, greatly degrading performance.

Namely, Dr. Dogfurry is not understanding how linear hashing works. The `next` bucket needs to be split when an overflow occurs in order to ensure that, on average, the #overflows is less than one.

-
- b. (2 points) Unlike extendible hash indexes, linear hash indexes can be implemented without the use of a directory.

What are the advantages of this?

It saves one I/O per search, since no directory page need be read. Of course, on extensive use, the directory pages are often in the buffer pool, so this advantage is marginal.

-
- c. (3 points) Even though linear hash indexes can be implemented without the use of a directory, there are some disadvantages to this. (Thus, they can also be implemented using a directory.)

Name two distinct disadvantages of the directory-less implementation.

- i. For the directory-less implementation, as the textbook describes, the buckets must be allocated at initialization, since they need to be physically sequential (or have sequential addresses). However, this is at odds with the fact that the #buckets grows over time. Whenever a new bucket would exceed the present allocation, a re-allocation would have to be done and everything copied.*
- ii. As seen in 5a, buckets can be empty in a linear hash. With a directory, a new bucket page need not be allocated until there is a record for it. Without a directory, we must allocate the page regardless.*

Okay. Many people interpreted this question as, "What are the disadvantages of linear hashes compared with extendible hashes." I accepted that interpretation. In the presence of skew, linear hashes will build up long overflow pages. Extendible hashes (generally) do not have overflow pages. So accesses through an extendible hash compared with through a linear hash in this case will be more efficient. Also under skew, linear hash can have low occupancy in many buckets, leading to worse space usage.

(Scratch space.)

(Scratch space.)

(Scratch space.)