# CS372H: Spring 2009 – Final Exam

## Instructions

- This exam is closed book and notes with one exception: you may bring and refer to a 1-sided 8.5x11-inch piece of paper printed with a 10-point or larger font. If you hand-write your review sheet, the text density should not be greater than a 10-point font would afford. For reference, a typical page in 10-point font has about 55-60 lines of text.
- If a question is unclear, write down the point you find ambiguous, make a reasonable interpretation, write down that interpretation, and proceed.
- State your assumptions and show your work. **Write brief, precise, and legible answers.** Rambling brain-dumps are unlikely to be effective. **Think before you start writing** so that you can crisply describe a simple approach rather than muddle your way through a complex description that "works around" each issue as you come to it. **Perhaps jot down an outline** to organize your thoughts. And remember, a **picture can be worth 1000 words.**
- For full credit, show your work and explain your reasoning.
- To discourage guessing and rambling brain dumps, I will give approximately 25% credit for any problem left *completely blank* (e.g., about 1.5 for a 5 or 6 point question, 2.0 for a 7 point question, 3.5 for a 14 point question)

  If you attempt a problem, you will get between 0% and full credit for it, based on the merits of your answer.

  Note that for the purposes of this rule, a *problem* is defined to be any item for which a point total is listed. *sub-problems* are not eligible for this treatment – if you attempt to answer any subproblem for a problem, you may as well attempt to answer all subproblems.
- Write your name on this exam

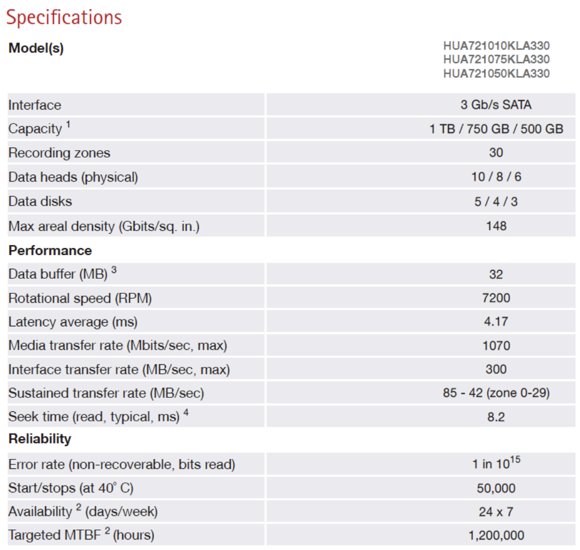
## 1 IO performance

Suppose a machine has 8GB of DRAM that it can use as a cache. Suppose the machine also has a 1TB disk of the kind specified to the right of this text and that this disk stores 128 GB ($2^{37}$ bytes) of data. In particular there are $2^{30}$ data records, each record containing an 8-byte key and a 120 byte value ($128 = 2^7$ bytes per record).

The records are stored in a tree. Each *leaf* is a 128KB ($2^{17}$) block of data stored contiguously on disk. In order to accommodate new record inserts, initially each leaf block is half full, so each initially contains 512 ($2^9$) records. Thus, there are $2^{21}$ leaf blocks on disk. Assume that these leaf blocks are stored in key-sorted order contiguously on some 256 GB ($2^{38}$ byte) region of the disk.

We'll call the leaf nodes level 0 (L0) of the tree. Each leaf's parent is a level 1 (L1) node, and so on.

The internal and root nodes (level $l$ nodes for $l > 0$) of the tree are 4KB blocks, each stored contiguously on disk. Each internal node is an array of records: *key, addr* indicating the disk address *addr* where the 128KB leaf block whose earliest key is *key* is stored. Assume that all nodes for any given level $l$ of the tree are stored contiguously on disk in sorted order.



Specifications

| Model(s) | HUA721010KLA330<br>HUA721075KLA330<br>HUA721050KLA330 |
|---|---|
| Interface | 3 Gb/s SATA |
| Capacity [1] | 1 TB / 750 GB / 500 GB |
| Recording zones | 30 |
| Data heads (physical) | 10 / 8 / 6 |
| Data disks | 5 / 4 / 3 |
| Max areal density (Gbits/sq. in.) | 148 |
| **Performance** | |
| Data buffer (MB) [3] | 32 |
| Rotational speed (RPM) | 7200 |
| Latency average (ms) | 4.17 |
| Media transfer rate (Mbits/sec, max) | 1070 |
| Interface transfer rate (MB/sec, max) | 300 |
| Sustained transfer rate (MB/sec) | 85 - 42 (zone 0-29) |
| Seek time (read, typical, ms) [4] | 8.2 |
| **Reliability** | |
| Error rate (non-recoverable, bits read) | 1 in $10^{15}$ |
| Start/stops (at 40° C) | 50,000 |
| Availability [2] (days/week) | 24 x 7 |
| Targeted MTBF [2] (hours) | 1,200,000 |

- How many levels of tree do you need to index this full data set? (7 points)

**Solution:** Each index record is 8 byte key + 8 byte addr (I assume 8 bytes to index large disks...). So, we can store $2^{12}/2^4 = 2^8$ index records per block. So, we need $2^{21}/2^8 = 2^{13}$ index blocks at L1, $2^5$ index blocks at L2, and 1 index block at L3 (the root).

- Assuming you want to provide high throughput for a workload that randomly accesses records in the leaves. Is it a reasonable engineering decision to cache all of the internal nodes of the tree in their entirety? For full credit, calculate the size of these nodes *and* make a quantitative engineering argument for whether caching this amount of data is a reasonable thing to do in this system. (7 points)

**Solution:** The total index size is $2^{13} * 2^{12} = 2^{25}$ bytes (32 MB) for level 1 plus some change ($2^5 * 2^{12} = 2^{17}$ = 128KB for level 2, and 4KB for level 3) for the other levels.
1GB of memory costs about \$20 and a disk costs \$100, so we're spending about \$.60 to cache the index for a \$100 disk. Increase the cost of the system by less than 1% to do so.
X points for getting the size right. X for expressing it compared to modern DRAM sizes. X for also looking at the cost of the disk.

Suppose a workload generator generates a long series of $2^{30}$ random record insert operations *INSERT key value*. Notice that each insert operation requires a read-modify-write of a leaf block.

- Estimate the steady state throughput for this workoad assuming a *synchronous update in place* strategy in which update $i$ is completed before update $i + 1$ is begun, and assuming that none of the leaf blocks become full during the run. (7 points)

**Solution:** We have to do a leaf read-modify-write for each request. We can cache 8GB of leaves, and that lets us avoid the read for 8/128 = 6.25% of the updates.
The read takes a random seek of 8.2ms (let's assume we store everything on the outer 1/3 of the disk and that the seek therefore only takes 8.2/3 = 2.7ms) and then 1/2 rotation (4.17ms) then a 64KB read (64KB/75MB/s = 1ms) = 2.7 + 4.17 + 1 = 7.97ms
If the read was cached, a write takes about the same.
If the read was not cached, a write just takes 1 rotation 8.34
So we have .9375 * (7.97 + 8.34) + .0625 * (7.97) = **15.79ms per update**

- Design a different update strategy that maximizes throughput (7 points)

**Solution:** Buffer 7GB of updates, then apply them in key-sorted order to minimize disk seeks to read-modify-write. In particular, read 1GB of updated leaf records from disk then write 1GB of updated leaf records to disk; repeat until all leaf records have been updated for a 7GB set of updates. Then repeat until all insert operations have been processed.

- Estimate the throughput your solution achieves (7 points)

  **Solution:** a 7GB batch of updates is 58.7M updates; there are 2 M leaf blocks, so we'll assume that every leaf block is updated at least once in each batch.
  Transfer rate is 85-42MB/s depending on which track we're at. I'll assume data are stored in the outer 1/3 of the tracks and that I get 70-85 MB/s for those tracks and average 75MB/s. So, reading or writing 1GB will take 13.33 seconds, reading then writing takes 26.66 seconds, and we need to read then write 256 1GB regions.
  Total time to apply each batch is 6825 seconds. Each batch includes 58.7M updates so we get a throughput of **8604 updates per second** or .1ms per update. Over 100x speedup.
  BTW, for total time: We have 128 GB / 7 GB = 18.3 batches; assume the .3 bath still updates each block, so we have 19 batches Total time 19 * 6825 = 124,800 seconds. About 1.5 days.

## 2 Short answer

- In the context of encryption protocols, what is the *nonce verification* rule? (5 points)
  - What conditions must hold to allow a node to apply the rule?

    **Solution:** node A believes node B once said MSG and MSG (or some part of MSG) is fresh
  - What conditions hold after the rule is applied?

    **Solution:** node A believes node B believes MSG
  - Explain what the rule means and why such a rule is needed.

    **Solution:** If I can apply nonce verification then I can know that the message is part of the current instance of the protocol instead of a message replayed from some other instance.
- Two-factor authentication is an alternative to passwords. Give an example of two-factor authorization and explain its advantages compared to passwords. (5 points)

  **Solution:** Each user has a smartcard and a password; to login to a machine, I insert my smart card and enter my password. The advantage is that an attacker cannot log in by guessing my password or my stealing my smart card – they must do both
- In the context of file systems, what is a *hard link*? (5 points)

  **Solution:** A directory entry comprising a name and a file ID is a hard link. Notice that multiple directory entries (with different names and/or in different directories) can link to the same file ID.
- Why is it harder to implement hard links in the DOS/Windows FAT file system than in the Unix Fast File System (FFS)? (5 points)

  **Solution:** FFS has an inode that can contain not just pointers to data but also per-file data structures like permissions, owner, and reference count. A FAT file ID refers to an entry in the FAT table, which is the first of a list of pointers to file blocks. There is no good place to store this metadata except in the directory entry, itself, but if we have multiple directory entries that refer to the same file, how do we keep them all consistent?

- Define the ACID properties of a transaction (give the name and definition of each) (5 points)

  **Solution:**

- I am running a web service on a single web server machine. My users are observing slow response time to their requests. I split my users so that half send their requests to the original machine and half send their requests to an identical second machine. My users' response time gets better by *much more* than a factor of two. What is a likely reason for such a *superlinear speedup* (5 points)

  **Solution:** Maybe the machine was thrashing (paging to disk), so the extra memory from the second machine made the whole workload fit in memory. Maybe the machine was heavily loaded so that queuing delays were large and the second machine reduced load to a point where queuing delays were small

## 3 Distributed commit

A commit protocol attempts to get all nodes to agree on a value. We are most interested in solving this problem assuming an *asynchronous fair network*.

An asynchronous fair network assumes that nodes may be arbitrarily slow, that the network may drop messages, reorder messages, or arbitrarily delay messages. However, it assumes that if a node periodically resends a message until it knows that the destination has received the message, then the message will eventually be received by the destination.

- Engineers have to make reasonable trade-offs, and it may sometimes be reasonable to ignore risks if they are really rare. For example, although assuming that the network is asynchronous might make sense for the Internet, perhaps in a machine room it is overly conservative. In particular, suppose I have a cluster of 10 machines, each machine has a 1 Gbit/s full-duplex Ethernet link, all Ethernet links go directly to a switch, the switch is extremely fast and has plenty of buffer space to queue packets instead of dropping them, and no machine ever sends at a rate higher than 1 Mbit/s. In such a system, packets should never be dropped and should arrive within a milliseconds of being sent, so rather than assuming no bound on message delivery, perhaps I could assume (conservatively?) that with an appropriate retransmission protocol, a packet is always received within 10 seconds of being sent unless the receiver has crashed. I could then design my protocol to work assuming a synchronous network rather than an asynchronous one.

  Why might it be dangerous to make such an assumption? (Be specific) (7 points)

  **Solution:** In normal operation, this should be OK, but what if...(a) my machine is really slow for a few minutes (e.g., updating virus files), (b) the network switch reboots, (c) a node reboots, (d) someone unplugs a cable, (e), ...

- Suppose that two nodes, $A$ and $B$, each store a local value ($value_A$ and $value_B$). Design a protocol that, under the asynchronous fair network model, guarantees that they first agree on the value *max* = MAX($value_A$, $value_B$) and then simultaneously print the value *max* to their screens *assuming that neither node crashes.* (7 points)

  **Solution:** No such protocol exists. Impossible.

Not only do we want to solve agreement assuming an asynchronous fair network, we also want some fault tolerance. For example, we want to guarantee that the protocol works even if $f$ nodes permanently crash.

Recall the 7-line nonblocking fault-tolerant consensus algorithm that required $n$ = 3f+1 nodes (e.g., 4 nodes to tolerate f = 1 crash.) Initially each node gets to unilaterally vote for RED or BLUE, but in subsequent rounds a node's vote depends on what other nodes vote for so that eventually all nonfaulty nodes settle on the same value.

Here is the pseudo-code for a node

```
$e = 0$;                      // election number
$c$ = RED or BLUE;            // vote this election
while (1){
    $e = e + 1$;
    send (VOTE, $e$, $c$) to all  // In background, keep periodically
                                  // sending to node i until I receive i's
                                  // vote for round $e$+1
    $VOTES$ = receive (VOTE, $e$, RED or BLUE) from 2f+1 nodes (including self) for election e
    $c$ = MAJORITY($VOTES$)
}
```

To determine the consensus value, I ask the nodes to tell me their current $c$ values for a particular round and wait until I receive n-f = 3 replies. If all three replies match, then the system has reached consensus on the value in that reply, and any subsequent read would return the same value. If the replies don't match, consensus has not been reached, and I can check again later.

Modify above algorithm to work with a COMMIT/ABORT voting rule instead of majority RED/BLUE voting. In particular, your solution should guarantee five properties

1. Nontriviality – either COMMIT or ABORT may be decided
2. Safety – the system decides COMMIT only if all nodes initially vote COMMIT (as in two-phase commit, the system may decide ABORT in a range of situations.)
3. Consensus – eventually all non-crashed nodes decide the same thing and once the system has reached a decision the decision does not change
4. Visibility – I can learn the final decision (or determine that no decision has yet been reached) by querying the state of any $n - f$ live nodes
5. Eventual liveness – assuming the network is an asynchronous fair network and that nodes automatically resend messages until they are known to be received, then eventually all non-crashed nodes agree on some value

- State your protocol for nonblocking COMMIT/ABORT agreement (7 points)

**Solution:** Call my initial vote my round 0 vote. I can vote either COMMIT or ABORT unilaterally.
In round 1 I will vote COMMIT only if I receive $n$ COMMIT votes from the initial "free choice" round 0
If I timeout before hearing from all n nodes or I hear anyone vote ABORT, then I vote abort in election 1
For all subsequent elections, I do majority rules

```
$e = 0$;    // election number
$c$ = COMMIT or ABORT;    // vote this election
while (1){
    $e = e + 1$;
    broadcast ($e$, $c$) to all
    $VOTES$ = receive ($e$, $vote$) from 2f+1 nodes (including self)
    if $e == 1${
        wait for up to TIMEOUT for f more votes
        if I received $n$ COMMIT votes
            $c$ = COMMIT
        else
            $c$ = ABORT
    else
        $c$ = MAJORITY($VOTES_a$)
}
```

Notice that I can't vote COMMIT in round 1 if I hear n-f COMMITS because the missing node might have voted abort
Notice that in rounds after round 1 I can't wait for all n messages because a node might have crashed
given that for later rounds, I must base my decision on n-f votes,
Notice that I can't simply vote ABORT in rounds after round 1 if I hear from only n-f nodes in later rounds because we might already have decided COMMIT
Notice that I can't simply vote ABORT (or COMMIT) in rounds after round 1 if I hear a single ABORT (or COMMIT) decision, because that would not be stable

- Prove that your protocol works by proving that it guarantees each of the 5 required properties (14 points)

*(contd)*