1. [20 points] Locking.

   a. [5 points] Why do we need the SIX lock?

   The SIX lock is useful for transactions that read a lot of records in a file and update a few. Without SIX, the options would be X on the whole file (low concurrency) or IX with individual S and X locks (high overhead.)

   b. [5 points] With which other locks is it compatible?

   Only with IS.

   c. [5 points] Would it make sense to hold an SIX lock on a record? Why or why not?

   This would only make sense (and even then it would be doubtful) if the record were not the lowest granularity of locking – for example, if you had field-level locking. If record level locking is the lowest granularity, SIX would buy you nothing over normal S and X locks.

   d. [5 points] Suppose that at some point we want to upgrade an SIX lock that we currently hold on a file to an X lock. Where should this upgrade request go in the queue of requests for locks on this file?

   The upgrade to X should go immediately after the current “granted group,” which would include the SIX and possibly some number of IS locks.

2. [5 points] Oracle CC.

   In Multiversion CC with timestamps, multiple versions of a data item are stored on a Rollback segment. When is it safe to delete old versions of an item?

   It is safe to delete a version when all active transactions have timestamps greater than the timestamp on the version in question.

3. [10 points] Degrees of consistency.
Consider the following pair of statements:

(i) Degree two consistency allows schedules not allowed by two-phase locking.
(ii) Two phase locking allows schedules not allowed by degree two consistency.

Which of these two is true? Explain your answer.

OK, this one was a nasty question. Both are true – note that this is about “2PL” not “strict 2PL.” Since degree two holds long write locks and short read locks, schedules that release write locks before commit are not possible, but they are with 2PL, so (ii) is true. Since 2PL cannot re-acquire a lock after releasing a lock, but degree two can, (i) is also true.

4. [10 points total] Consider the B-link algorithm. During its operation index pages are not shared and updated in an arbitrary way. So what assumptions does it require on how index pages are read and updated?

The important thing here is that on updating, the algorithm gets write locks, and it atomically reads the page into private storage, atomically writing it back when it is finished. This means that writes cannot conflict, and reads always either see a completed write or they don’t see the write at all! Optimistic Concurrency Control [15 points]

Using the Kung and Robinson optimistic concurrency control algorithm, give an example schedule (that is, a sequence of reads, writes, and validations) in which there are three transactions T1, T2, and T3, and where T1 causes T2 to be invalidated but then T1 is itself invalidated by T3.

Here the idea was to use the parallel validation protocol. Suppose T1 and T3 both write X, and T2 also reads X. Then suppose T3 starts validating (enters the “active” set) then gets “swapped out.” Next T1 begins validating but is swapped out. Finally, T2 hits validation. Since the write set of T1 intersects the read set of T2, T2 will fail (be invalidated by T1.) Next T1 starts up again, sees that its write set conflicts with T3’s, so it fails (it is invalidated by T3).

5. [15 points] ARIES.

a. [8 points] ARIES is said to “redo” history. What does this mean, and why does it do this?

ARIES is said to “redo” history because after a crash, it redoes the actions of all transactions up to the point of the crash, including failed transactions (those that didn’t commit.) So it redoes what actually happened, not just the actions of successful transactions. This helps out in allowing for CLRs during recovery (and hence the bounded space guarantee), and also helps with enabling logical redo/undo.
b. [7 points] The paper makes the claim that ARIES uses bounded space for recovery even in the presence of repeated crashes. Explain why this is true.

ARIES only writes log records during the UNDO phase. It will write at most one log record per undo (due to the use of CLRs) so no matter how many crashes occur, the space used during recovery is bounded.

6. Transaction Management in R* Distributed Database Management System [15 points]

The three versions of two-phased commit that we discussed in class all have the goal of “forgetting” transactions as soon as possible. Explain what this means, and for each version of two-phased commit, state when the coordinator can “forget” a transaction, and say (briefly!) why this is correct.

This one was hard to answer concisely! The main ideas were:

- “forgetting” a transaction means deleting it from the in-memory data structure maintained by the coordinator. This is safe whenever the coordinator “knows” that if any transaction asks it about the transaction, with no knowledge of the transaction it will give the correct answer.

- Standard 2PC: after receiving “ack” from all subordinates it is safe to forget, because the coordinator knows it will never be asked about that transaction again.

- Presumed Abort: in the commit case, it is safe to commit after receiving all acks from subordinates, because they will not ask again. In the abort case, it can forget earlier – certainly after sending out “aborts” after receiving one “no” vote (it would be hard to forget earlier, how could you even send out the aborts?)

- Presumed Commit: in the abort case, it is safe to commit after receiving all acks from subordinates. For the commit case, it is safe to forget after forcing the “commit” record to disk.

6. Join Algorithms [10 points]

Describe the symmetric hash join algorithm and argue why it produces the correct answer.

Symmetric hash, as described in class, assumes both R and S fit in memory. It works as follows. Begin scanning both relations concurrently. When processing a tuple of R, insert
it into an R hash table, and probe the S hash table; when processing a tuple of S, insert it into the S hash table and probe the R hash table.

For correctness, assume a tuple <r,s> should be in the answer to the join. If the r tuple is processed first, then it will not find s, and will be inserted into the R hash table. When s arrives, it will find r in the r hash table, so <r,s> will be generated. The reverse (s arriving before r) is symmetric. So each answer tuple is generated exactly once.