1) Instead of having just one copy of our ACID database on one machine, we replicate it across three machines. What happens to our performance, in general?

If we have an ACID database on three machines and we want to read something from the database, then we can send one read request to each machine and we only need to wait for the fastest machine to respond, so in general, **reads are faster**. On the other hand, for a write to complete, it needs to be replicated across all machines, so we need to wait for the slowest machine to respond, and thus **writes are slower**.

2) Suppose that you are a Coordinator node, and you're participating in a transaction with 7 Participants. How many YES votes do you need to COMMIT?

We need every single participant to vote YES, so we need to wait for **7** YES votes.

3) You wake up. You have no memory of what happened. You're a Participant node, and you realize that you've just crashed. You look at your logs and see that the last record that appeared on transaction T1 is a PREPARE record. What do you do?

We wrote PREPARE, so there's still a chance we might COMMIT later in the transaction. On the other hand, we can't COMMIT yet because some other node might have voted NO and thus we must abort. Therefore, we have to ask the **Coordinator for its status and wait**; the Coordinator recovery process will later return us the result of the commit (COMMIT or ABORT).

4) Suppose we have a Coordinator and 7 Participants. When *could* the Coordinator ABORT a transaction?

- **The Coordinator has written ABORT in its logs**

Clearly, if the Coordinator wrote ABORT, we're aborting the transaction.

- **The Coordinator sent PREPARE to all Participants and received 6 YES votes but hasn't yet heard from the seventh node**

In this case, the Coordinator may decide that the seventh node crashed and ABORT, or it can choose to keep waiting and hope that a YES arrives.

- The Coordinator sent COMMIT to all Participants and received 6 ACKs but hasn't yet heard from the seventh node

If the Coordinator sends COMMIT at any point, it must commit!

- **A Participant has written ABORT in its log**

If a participant has written ABORT in its log, that either means it voted NO (so we abort) or it received an ABORT message from the Coordinator (in which case the Coordinator decided to abort, so we do too).

- A Participant has written COMMIT in its log

If a Participant wrote COMMIT in its log, we are definitely going to commit! This only happens when the Coordinator received all YES votes and chose to COMMIT, sending out the commit message after the log entry was flushed to disk.

- **A Participant has voted YES**

All Participants must vote YES to commit. It's possible that some other participant has voted / will vote NO.

4) Suppose we have a Coordinator and 7 Participants. When *could* the Coordinator COMMIT a transaction?

- The Coordinator has written ABORT in its logs

Clearly, if the Coordinator wrote ABORT, we're aborting the transaction.

- **The Coordinator sent PREPARE to all Participants and received 6 YES votes but hasn't yet heard from the seventh node**

In this case, the Coordinator may decide that the seventh node crashed and ABORT, or it can choose to keep waiting and hope that a YES arrives.

- **The Coordinator sent COMMIT to all Participants and received 6 ACKs but hasn't yet heard from the seventh node**

If the Coordinator sends COMMIT at any point, it must commit!

- A Participant has written ABORT in its log

If a participant has written ABORT in its log, that either means it voted NO (so we abort) or it received an ABORT message from the Coordinator (in which case the Coordinator decided to abort, so we do too).

- **A Participant has written COMMIT in its log**

If a Participant wrote COMMIT in its log, we are definitely going to commit! This only happens when the Coordinator received all YES votes and chose to COMMIT, sending out the commit message after the log entry was flushed to disk.

- **A Participant has voted YES**

Of course, we have to wait for all Participants to vote YES before we commit.

6) Which of the following operators are monotonic? In other words, which of these operators will cause data to be eventually consistent on a distributed database?

Note: we decided that this question wasn't adequately covered in class, and we'll be awarding full points for any attempt.

For an operation to be monotonic / eventually consistent, it must be commutative, associative, and idempotent. Addition and XOR are not idempotent: for example, (x + y) + (y + z) is not x + y

+ z. Set difference is not commutative: S - R is not R - S (as covered in lecture). Thus, the only monotonic operator is **min(x, y)**.