KEVIN JOHNSON

MODULE 9 READING COMPREHENSION

1. What does ACID stand for? And what does each part mean? (2 Pts)

- Atomicity: ACID atomicity describes what happens if a client wants to make several writes but a fault occurs after some of the write have been processed. If the writes are grouped together into an atomic transaction, and the transaction cannot be completed due to a fault, then the transaction is aborted and the database must discard or undo any writes it has made so far in that transaction. If a transaction is aborted, the application can be sure that it did not change anything, so it can be safely retried.
- Consistency: The idea of ACID consistency is that you have certain statements about your data (invariants) that must always be true. If a transaction starts with a database that is valid according to these invariants, and any writes during the transaction preserve the validity, then you can be sure that the invariants are always satisfied.
- Isolation: Isolation in the sense of ACID means that concurrently executing transactions are isolated from each other. Each transaction can pretend it is the only transaction running on the entire database. The database ensures that when the transactions have committed, the result is the same as if they had run serially, even though in reality they may have run concurrently.
- Durability: Durability is the promise that once a transaction has committed successfully, any data it has written will not be forgotten, even if there is a hardware fault or the database crashes.

2. What are dirty reads and dirty writes? (2 Pts)

The most basic level of transaction isolation is read committed. It makes two guarantees:

1. When reading from the database, you will only see data that has been committed (no dirty reads).
    - This means that any writes by a transaction only become visible to others when that transaction commits. There are a few reasons why it is useful to prevent dirty reads:
        1. If a transaction needs to update several objects, a dirty read means that another transaction may see some of the updates but not others.
        2. If a transaction aborts, any writes it has made need to be rolled back.
2. When writing to the database, you will only overwrite data that has been committed (no dirty writes).
    - Dirty writes are when a later write overwrites and earlier written value that has yet to be committed. Transactions running at the read committed isolation level must prevent dirty writes, usually by delaying the second write until the first write's transaction has committed or aborted.

3. What is read skew and write skew? (2 Pts)

Read skew occurs when during the course of a transaction, a row is retrieved twice and the values of the row change between reads. For example, if User A runs a query twice and between the two queries, User B runs a transaction and commits. The row that User A has queried will now have a different value. Certain situations cannot tolerate these inconsistencies, such as backups and analytic queries. Snapshot isolation is the most common solution to this problem. Snapshot isolation reads from a consistent snapshot of the database, meaning the transaction sees all the data that was committed in the database at the start of the transaction. Even if the data is subsequently changed by another transaction, each transaction sees only the old data from that particular point in time.

In a write skew, two transactions (T1 and T2) concurrently read an overlapping data set (values V1 and V2), concurrently make updates (T1 updates V1, T2 updates V2), and concurrently commit, having never seen the update performed by the other. Examples of write skew include meeting room booking systems, multiplayer games, claiming a username, and preventing double-spending. The solutions to write skew are much more restrictive. Some databases allow you to configure constraints, which are then enforced by the database. If you cannot use a serializable isolation level, the second-best option is to explicitly lock the rows that the transaction depends on.

4. What is serializable isolation? (2 Pts)

Serializable isolation is an optimistic concurrency control technique. Instead of blocking if something potentially dangerous happens, transactions continue anyway, in the hope that everything will turn out al right. When a transaction wants to commit, the database checks if anything happens. If so, the transaction is aborted and has to be retried. Only transactions that executed serializably are allowed to commit. Serializable isolation is based on snapshot isolation, meaning all reads within a transaction are made from a consistent snapshot of the database. On top of snapshot isolation, serializable isolation adds an algorithm for detecting serialization conflicts among writes and determining which transactions to abort.

5. What is Two-Phase Locking and how is it used? (2 Pts)

In two-phase locking, several transactions are allowed to concurrently read the same object as long as nobody is writing to it. As soon as anyone wants to write an object, exclusive access is required:

- If transaction A has read an object and transaction B wants to write to that object, B must wait until A commits or aborts before it can continue.
- If transaction A has written an object and transaction B wants to read that object, B must wait until A commits or aborts before it can continue.

Two-phase locking is a pessimistic concurrency control mechanism that is based on the principle that if anything might possibly go wrong, it is better to wait until the situation is safe again before doing anything.

In two-phase locking, the blocking of readers and writers is implemented by having a lock on each object in the database. The lock can either be in shared mode or in exclusive mode. The lock is used as follows:

- If a transaction wants to read an object, it must first acquire the lock in shared mode. Several transactions are allowed to hold the lock in shared mode simultaneously, but if another transaction already has an exclusive lock on the object, these transactions must wait.
- If a transaction wants to write to an object, it must first acquire the lock in exclusive mode. No other transaction may hold the lock at the same time, so if there is any existing lock on the object, the transaction must wait.
- If a transaction first reads and then writes an object, it may upgrade its shared lock to an exclusive lock. The upgrade works the same as getting an exclusive lock directly.
- After a transaction has acquired the lock, it must continue to hold the lock until the end of the transaction.