Relational DBMS and NoSQL Databases Study Guide

Relational Database Transactions and ACID Properties

Transaction Definition: A **transaction** is a logical unit of work consisting of one or more database operations (such as reads and writes) that are executed as a single unit. In SQL, transactions typically begin with `START TRANSACTION` and end with either a **COMMIT** (successful completion) or a **ROLLBACK** (abort). For example, transferring $50 from Account A to Account B can be treated as one transaction: deduct $50 from A and add $50 to B. If any part of this sequence fails, the entire transaction is rolled back so the accounts remain consistent.

ACID Properties: Transactions in a relational DBMS are expected to exhibit **ACID** properties:

- **Atomicity:** All operations in a transaction succeed as a whole or fail as a whole. The system will not apply a portion of the transaction – **either the entire set of changes happens or none do**. This guarantees that partial updates do not occur. For instance, in the account transfer example, it's not possible to deduct money from Account A without adding it to Account B; either both steps happen or neither does.

- **Consistency:** A transaction should take the database from one consistent state to another. This means the defined integrity rules (constraints) are preserved. If each transaction enforces the correctness of data (e.g., no rule violations), the database remains consistent. For example, if a rule says total debits must equal total credits, that rule should hold true before and after each transaction.

- **Isolation:** Each transaction executes as if it were the only one running in the system. Even if transactions run concurrently, isolation ensures intermediate results of one transaction are not visible to others. In practice, the effects of other ongoing transactions are not seen until those transactions are committed. This prevents *dirty reads* or interference among operations.

- **Durability:** Once a transaction commits, its effects are **permanent**, even in the face of subsequent system failures. The DBMS uses techniques like logging and backups so that committed changes survive power outages, crashes, or other failures. For example, after committing the account transfer, the new balances should persist (perhaps by being written to disk and logs) so they are not lost if the database crashes moments later.

Example – ACID in Action: Consider a transaction that transfers $50 from Alice's account (A) to Bob's account (B):

pgsql
CopyEdit
```
START TRANSACTION;
    UPDATE Accounts SET balance = balance - 50 WHERE account_id = 'A';
    UPDATE Accounts SET balance = balance + 50 WHERE account_id = 'B';
COMMIT;
```

Atomicity ensures both updates happen together or not at all. Consistency ensures the total balance A+B remains the same (other business rules intact). Isolation ensures if another concurrent transaction is looking at accounts, it won't see an intermediate state where A is deducted but B not yet credited. Durability ensures that once committed, the transfer is not lost.

Practice Questions: Transactions and ACID

- Define "transaction" in the context of databases. Why is it called a logical unit of work?
- Explain each of the ACID properties in your own words and give a real-world analogy or example for each.
- What happens if a transaction fails midway and why is atomicity crucial in that case?
- A system crashes right after a transaction commits. Which ACID property ensures the changes are not lost, and how might the DBMS implement this?

Concurrency Control and Serializability

When many users and applications access a database at the same time, their transactions can interleave. **Concurrency control** is the process of managing simultaneous operations on the database without letting them interfere with each other. Without proper control, interleaved execution of transactions can lead to several anomalies that violate consistency:

- **Lost Update:** Two transactions read the same data and both make updates, but one update is lost because it was overwritten by the other. For example, if T1 and T2 both read a bank balance of $100 and each subtract $10, each might write back $90, but one of the $10 deductions will overwrite the other, resulting in $90 instead of the correct $80. This problem is avoided by preventing one transaction from overwriting data that another transaction has updated but not yet committed.

- **Uncommitted Dependency (Dirty Read):** One transaction (**T2**) reads data that another transaction (**T1**) has written but not committed. If T1 rolls back, T2 has read data that never officially existed. For instance, T1 temporarily raises a balance from $100 to $200, and T2 reads this $200. If T1 aborts, the balance returns to $100, but T2 might have used the $200 value in further calculations, causing inconsistency. The solution is to not allow transactions to read uncommitted changes of others.

- **Inconsistent Analysis (Nonrepeatable Read):** A transaction reads several values (e.g., summing across accounts) while another transaction is updating some of those values. The first transaction sees an inconsistent snapshot – some data from before the second transaction's update and some after. For example, T6 is summing balances of accounts X, Y, Z while T5 transfers $10 from X to Z; T6 might read X before the transfer and Z after, leading to a total that is $10 off. Proper isolation (e.g., locking or versioning) prevents seeing a partial update.

- **Phantom Read:** A special case of inconsistency where a transaction re-executes a query and finds rows that weren't there before ("phantom" rows) because another transaction inserted them. (This is typically controlled by higher isolation levels or range locks to prevent new rows from appearing in a query's result set during a transaction.)

Serial and Serializable Schedules: The simplest way to avoid interference is to run transactions one after another (a **serial schedule**), but that sacrifices concurrency. Instead, we aim for **serializability**, which allows transactions to interleave as long as the outcome is the same as *some* serial execution of those transactions. In a **serializable schedule**, even though operations are interleaved, the net effect on the database is equivalent to executing the transactions sequentially in some order.

- Two schedules are considered **equivalent** if each transaction's outcome (and the overall database state) is the same in both schedules. Serializable schedules avoid the anomalies listed above by guaranteeing consistency as if transactions ran one by one.

Conflict Serializability: A common criterion for serializability is **conflict serializability** – it focuses on pairs of operations that *conflict*. Two operations conflict if **(i)** they belong to different transactions, **(ii)** they access the same data item, and **(iii)** at least one of them is a write. For instance, a read of X in T1 and a write of X in T2 conflict, as do two writes to the same item. However, two reads of X do not conflict (no write involved), and a read of X in T1 and a write of Y in T2 do not conflict (different data). A schedule is conflict-serializable if we can swap non-conflicting operations and transform the schedule into a serial one. In practice, we use a **precedence graph** (or serialization graph) to test for conflict serializability: each transaction is a node, and a directed edge from T_i to T_j indicates that an operation of T_i precedes and conflicts with an operation of T_j. If this graph has no cycles, the schedule is conflict-serializable (equivalent to a serial order given by the graph).

Recoverability: Even if a schedule is serializable, it must also be *recoverable* to avoid violating transaction atomicity in the event of aborts. A **recoverable schedule** is one where if a transaction T2 reads data written by T1, then **T1 commits before T2** does. This ensures that T2 is not dependent on an uncommitted transaction. If a schedule is not recoverable (e.g., T2 commits based on a value written by T1 that later aborts), it can lead to an irreversibly inconsistent state.

- **Cascading Rollbacks:** If a schedule is recoverable but a transaction abort causes other transactions to abort in turn, this is a *cascading abort*. For example, T1 writes X, T2 reads X then commits, and later T1 aborts – T2 must also abort (too late if it committed) or be undone, causing a chain reaction. A **cascadeless schedule** avoids this by not allowing a transaction to read data from another *before* the writer commits, thereby preventing the need for cascading rollbacks. In a cascadeless schedule, *all reads are on committed data*. A stronger condition is **strictness**: a **strict schedule** is one where transactions do not read *or write* data modified by an uncommitted transaction. In strict schedules, any value written by a transaction is not even read by others until that transaction commits (and similarly, uncommitted data is not overwritten by others). **Strict schedules** are both cascadeless and simpler to recover, because if a transaction aborts, no other transaction has seen or altered its uncommitted changes.

Practice Questions: Concurrency and Serializability

- What is the difference between a serial schedule and a serializable schedule? Why do we allow nonserial schedules at all?
- Give an example of a lost update problem. How does concurrency control prevent it?
- Define conflict serializability. How would you determine if a given interleaved schedule is conflict-serializable (what tool or test is used)?
- Explain what a recoverable schedule is. Why is the schedule where T2 reads from T1 and commits before T1 commits problematic?
- What is a cascading abort? How do cascadeless schedules eliminate cascading aborts?
- **Challenge:** Draw a small precedence graph for two transactions that conflict and show how a cycle indicates a non-serializable schedule.

Locking Methods for Concurrency Control

One way to achieve serializability is through **locking**. A lock is a mechanism that prevents concurrent access to a data item by other transactions. The two primary lock modes are usually

Shared (S) for reading and **Exclusive (X)** for writing. A shared lock on an item allows other shared locks (multiple transactions can read the same item at once), but an exclusive lock prevents any other access (no other reads or writes) on that item

en.wikipedia.org

en.wikipedia.org
. Transactions follow a locking protocol to ensure consistency:

- **Two-Phase Locking (2PL):** This protocol requires that each transaction acquire all the locks it needs **before** releasing any lock. In the **growing phase**, the transaction may acquire locks (and not release any). Once it starts releasing a lock, it enters the **shrinking phase** and cannot obtain new locks. By locking all data before using and not acquiring new locks after releasing, 2PL ensures no cyclic dependencies of locks, thereby guaranteeing conflict serializability. In effect, 2PL forces transactions to lock in a way that is equivalent to some serial order.

- **Strict 2PL:** A commonly used variant of 2PL is *strict two-phase locking*, where a transaction holds **all** its exclusive locks until it commits (releases them only at commit time). This not only ensures serializability but also makes the schedule **strict** (as defined earlier), simplifying recovery (no other transaction will read uncommitted data).

Lock Granularity: Locks can be applied at different levels of granularity – e.g., the entire database, a table, a page (disk block), a tuple (row), or even a field (column value). Fine granularity (e.g., row-level locks) maximizes concurrency (many transactions can lock different rows in the same table) at the cost of overhead (more locks to manage). Coarse granularity (e.g., table-level locks) reduces overhead (one lock covers a large item) but can unnecessarily restrict concurrency (transactions lock data they don't actually need). For example, if one transaction needs **many** records from a table, it might be more efficient to lock the whole table (coarse lock) than lock hundreds of individual rows – but this prevents other transactions from accessing *any* row of that table in the meantime. Modern DBMSs often implement **multiple granularity locking** with a hierarchy (database > table > page > record) and use **intention locks** (like IS, IX) to allow mixed granularity locking safely

en.wikipedia.org

en.wikipedia.org
. Intention locks signal a transaction's intent to lock finer-grained items underneath a coarse item, so that, say, one transaction can exclusively lock a row in a table (having an intention lock on the table) while another transaction can still read other rows in that same table.

Deadlocks: The use of locks can lead to a situation called a **deadlock**, where two or more transactions are waiting indefinitely for each other to release locks. For example, T1 locks record A and T2 locks record B; then T1 tries to lock B and waits (because T2 has it), while T2 tries to lock A and waits (because T1 has it). Now, neither can proceed – this is a deadlock **cycle**. In general, if a cycle exists in the waits-for graph (a graph of transactions waiting for locks held by others), the transactions will be deadlocked. Deadlocks can be handled by the DBMS in a few ways:

- **Deadlock Detection:** The system can build a waits-for graph of transactions and their lock wait dependencies. If a cycle is detected, the system chooses a victim transaction to abort (rollback) to break the cycle, allowing others to proceed. The aborted transaction will be restarted later.

- **Deadlock Prevention:** The system imposes ordering constraints to prevent cycles from forming. For example, the *Wait-Die* and *Wound-Wait* schemes use timestamps to decide if a transaction should wait or abort when a lock conflict arises. In *Wait-Die*, an older transaction can wait for a younger one, but a younger must abort ("die") if it needs a lock held by an older. In *Wound-Wait*, an older will preempt (wound) a younger by forcing it to abort, whereas a younger must wait if a lock is held by an older. These rules prevent cycles by systematically aborting one of the two in a potential wait cycle.

- **Timeouts:** A simpler approach some systems use – if a transaction waits too long for a lock, assume it's deadlocked and abort it. This doesn't *guarantee* the wait was due to a deadlock, but is easy to implement.

Practice Questions: Locking and Deadlocks

- Explain how two-phase locking ensures serializability. What are the two phases and why is the protocol important?

- What is the difference between a shared lock and an exclusive lock? Give an example scenario for each.

- Consider a database where one transaction is updating a set of rows in a table and another is reading different rows of the same table. How could lock granularity affect their concurrency?

- Describe a deadlock scenario involving two transactions and two data items. How might the DBMS detect this situation?

- What is strict two-phase locking and how does it relate to recoverability?

- How do *Wait-Die* and *Wound-Wait* deadlock prevention strategies decide which transaction to abort?

Timestamp Ordering and Thomas's Write Rule

Locks are a **pessimistic** approach to concurrency (assuming conflicts will happen and preventing them by waiting). An alternative **timestamp ordering** approach is more optimistic – transactions proceed without locks, but if a conflict is detected based on timestamps, offending transactions are rolled back. Each transaction is given a unique **timestamp** (e.g., at start time). The DBMS also tracks two timestamps for each data item: a **read timestamp** (the largest timestamp of any transaction that read the item) and a **write timestamp** (the largest timestamp of any transaction that wrote the item). The basic **Timestamp Ordering Protocol** works as follows:

- **Rule for Read operations:** When transaction T (with timestamp ts(T)) tries to `READ(x)`:
    - If `ts(T)` is **less** than `write_timestamp(x)` (i.e. T started *before* the most recent write on x by another transaction), then T is too "old" – some younger transaction has already written x. Allowing T to read the older value would violate consistency (T missed the new value). In this case, **abort T** and restart it with a new timestamp (so it will come "later" next time).
    - Otherwise (T is newer than or as new as the last writer of x), allow the read to proceed. Update `read_timestamp(x)` to `max(read_timestamp(x), ts(T))`.
- **Rule for Write operations:** When T tries to `WRITE(x)`:
    - If `ts(T)` is **less** than `read_timestamp(x)`, it means x has been read by a younger transaction already. T is trying to write a value that some newer transaction has already seen (which would violate serializability, as that younger transaction assumed the value of x wouldn't change). **Abort T** (and restart with a new timestamp) in this case.
    - Else if `ts(T)` is **less** than `write_timestamp(x)`, then x has already been written by a younger transaction. T's write is obsolete – it's like T missed the latest update and is trying to write an outdated value. Under the *basic* protocol, this is also an **abort T** situation.
    - Otherwise, if T is newer than any who read or wrote x, the write can proceed. The system performs the write and sets `write_timestamp(x) = ts(T)`.

If a transaction is aborted due to timestamp ordering rules, it will be restarted with a new (typically larger) timestamp, ensuring progress (eventually it will be the youngest transaction and not get aborted for being too old). This protocol ensures **conflict serializability**: transactions' effective order is their timestamp order. Notably, because there are no locks, **deadlocks cannot occur** – transactions never wait, they only get aborted if they conflict and violate the order.

Thomas's Write Rule: The basic protocol above can be overly strict. Thomas's Write Rule is an optimization that relaxes the second condition for writes to avoid unnecessary aborts. Specifically, if `ts(T) < write_timestamp(x)` (T wants to write an item that has already been updated by a younger transaction), instead of aborting T, we **ignore (skip)** T's write operation

file-68mysc1gtiamp6esfipfho
. In other words, T's update is discarded as obsolete, but T can continue. This is safe because a younger transaction has already written a value for x; the value T was going to write is outdated and not needed. By *rejecting obsolete writes* rather than aborting the transaction, Thomas's rule allows more schedules to be accepted (some that are not conflict-serializable but are **view-serializable**). However, designers must be careful: while this improves concurrency, the resulting schedule might not be conflict-serializable (cannot be transformed by swapping non-conflicting operations into a serial order), but it **will be** view-serializable (the final outcome still corresponds to some serial execution in terms of final values). Thomas's rule is particularly useful in distributed databases and certain NoSQL stores that prefer availability; they may temporarily accept out-of-order writes knowing that obsolete ones can be ignored.

Practice Questions: Timestamps

- How do timestamps differ from locks in managing concurrent transactions? What potential advantage do they have regarding deadlocks?

- Transaction T1 has timestamp 5 and transaction T2 has timestamp 10. Data item X currently has a write-timestamp of 8 and read-timestamp of 9. If T1 attempts to read X, what will happen? If T1 attempts to write X, what will happen? Explain using the timestamp protocol rules.

- What does Thomas's Write Rule change about the basic timestamp ordering protocol? Why might a database choose to use Thomas's rule?

- In timestamp ordering, why is it necessary to assign a new timestamp to a restarted transaction after it aborts? What could go wrong if we reused the old timestamp?

- **Challenge:** Consider a scenario where T1 (ts=1) writes X, then T2 (ts=2) writes X, then T1 (still alive) tries to write X again. Describe how basic timestamp ordering handles this versus how it would be handled with Thomas's Write Rule.

Recovery in Database Systems

Even with careful concurrency control, databases can encounter failures that require **recovery** to preserve correctness. Recovery is about restoring the database to a consistent state after a failure, ensuring the **A**tomicity and **D**urability of transactions (the “AD” of ACID).

Why Recovery is Needed: Many types of failures can affect a database:

- **System Crashes:** A hardware, OS, or DBMS crash can cause loss of data in memory (the volatile buffer/cache) while the system is running. Transactions in progress (not yet fully saved to disk) will be partially executed.

- **Media Failures:** Disk failures can destroy or corrupt the persistent storage (database files). Portions of the database might become unreadable.

- **Application Errors:** Bugs or exceptions in the application (or in the DBMS) might terminate transactions abnormally and leave the database in an intermediate state.

- **Natural Disasters:** Fires, floods, power outages, etc. can damage data centers or hardware.

- **Human Errors or Sabotage:** Accidental data deletion or malicious actions can introduce incorrect data or remove data.

Without recovery mechanisms, such failures could lead to lost updates or an inconsistent database (violating atomicity or durability). For example, if a transfer transaction debited Account A, then the system crashed before crediting Account B, Account A’s money is effectively lost. Recovery procedures must undo the debit on A to restore consistency.

Transactions and Recovery: Transactions are the unit of recovery. The recovery manager keeps track of each transaction’s changes so that:

- If a transaction **commits**, all its changes are made durable (they will survive failures). If a crash occurs after a commit but before changes hit the disk, the system will **redo** that transaction’s updates during recovery.

- If a transaction **aborts** (or is incomplete at crash time), none of its changes should persist. The system will **undo** any effects of that transaction, whether it aborted on its own or was incomplete due to a crash.

The mechanism that makes this possible is the **transaction log** (also called the journal). The log is an append-only record of all important events in the database: transaction start, each write (often recording the before- and after-image of the data), commits, aborts, and checkpoints. As part of normal operation, the DBMS writes to the log **before** it writes to the database (this is the **Write-Ahead Logging (WAL)** protocol). WAL ensures that if the database crashes, the log contains enough information to recover:

1. No change is applied to the database on disk without its log entry being safely on disk. (This makes it possible to undo a partially applied change or redo an applied change after a crash.)

2. All log records for a transaction are flushed to disk before the transaction is considered committed. (This ensures durability – the commit won't be lost because the record of it is on stable storage.)

Recovery Facilities: A DBMS typically provides several facilities to assist recovery:

1. **Backup Mechanism:** Periodic backups of the database are taken (full dumps or incremental backups). In case of catastrophic failure (like a disk crash destroying the database), the backup can be restored first. For safety, backups are often stored off-site.

2. **Logging Facility (Journal):** As mentioned, the log records all transactions and changes. This is stored on stable storage (and often duplicated for safety). Using the log, the DBMS can *redo* or *undo* transactions as needed.

3. **Checkpointing:** A **checkpoint** is a point in time when the DBMS writes all in-memory modified data (dirty pages) to disk and also records a checkpoint entry in the log. A checkpoint typically notes “all transactions up to this point have their changes on disk”. Checkpoints speed up recovery by limiting how far back in the log the recovery process needs to look. After a crash, the system can start recovery from the last checkpoint rather than from the very beginning of the log.

4. **Recovery Manager:** The component of the DBMS that orchestrates the restore process after a crash. It reads the log, identifies which transactions need to be redone or undone, and applies the necessary fixes.

Recovery Techniques: There are a few fundamental approaches to recovery using the log:

- **Deferred Update (Deferred Commit):** In this approach, transactions do all their work in local memory or a temporary area and **do not write any changes to the database until they commit**. The log records the intended updates (after-images) but the database is updated only at commit time. If a transaction fails before committing, no changes have hit the database, so nothing needs to be undone (just ignore its log records). If a failure occurs (system crash), any transaction that had committed (and thus whose after-images might not yet be applied to the database) will be **redone** from the log during recovery. Deferred update ensures that **no undo is ever needed** after a crash (since uncommitted transactions never modified the DB). However, it requires that the system can apply all pending changes of committed transactions on restart.

- **Immediate Update:** Here, the database may be updated **before** the transaction commits (e.g., writing to the buffer cache, which may get flushed to disk even before commit). This is more complex to manage but allows changes to be written gradually. If a failure happens, the recovery process will **undo** any changes from transactions that did not commit (using the "before-image" recorded in the log) and **redo** the changes of transactions that did commit. Immediate update absolutely requires WAL – the log must have the info to undo/redo any operation that made it to disk. Most relational databases use immediate update with a combination of undo and redo during recovery. Typically, recovery involves an **analysis** phase (determine which transactions were active), an **undo** phase for losers (those never committed), and a **redo** phase for winners (committed transactions that might not have their changes safe on disk).

- **Shadow Paging:** This technique avoids logs (to some extent) by keeping multiple copies of data pages. When a transaction starts, it keeps a **shadow page table** (a copy of the current disk page table). Updates are made to new pages, and the current page table is updated to point to these new pages. The shadow page (which points to the original pages) is not changed during the transaction. If the transaction aborts, the current page table is thrown away, and the shadow (original) is used, so no changes persist. If the transaction commits, the system simply replaces the disk's master page table with the current page table in one atomic action. This makes transaction commit very fast (just swap page tables) and if a crash occurs mid-transaction, the shadow (unchanged) table is still in place, effectively rolling back changes. Shadow paging has overhead of copying pages and can be tricky with concurrent transactions, so it's less commonly used in modern systems than logging.

Example Recovery Scenario: Suppose a system crashed, and upon restart the recovery manager finds the following in the log (simplified):

arduino
CopyEdit
```
T1 start
T1 write A, before=80, after=60
T2 start
T2 write B, before=70, after=90
T2 write E, before=100, after=200
T2 commit
... (checkpoint) ...
T1 write D, before=112, after=115
T1 commit
T3 start
```

```
T3 write B, before=90, after=100
[CRASH]
```

After analyzing, the recovery process might determine: T1 and T2 committed; T3 did not commit by the time of crash. Using the log, it will **redo** T1 and T2's changes (to ensure A, B, D, E reflect the after-values from those commits) and **undo** T3's change to B (since T3 didn't commit, B should be restored to 90). Because of checkpoint, it may skip over parts of T1 and T2 if they were included in a checkpoint prior to crash.

Practice Questions: Recovery

- List three types of failures that a database must be able to recover from. How might each of these failures affect transactions in progress?
- What is the purpose of the log in database recovery? Why must the log be written to stable storage *before* the actual data is written?
- Explain the difference between deferred update and immediate update recovery techniques. What does each one require in terms of undo or redo?
- Describe what a checkpoint is and how it helps minimize recovery time.
- In the event of a crash, how does the recovery manager determine which transactions to undo and which to redo?
- **Challenge:** Consider a scenario with two transactions T1 and T2 where T1 commits before the crash and T2 does not. If the system uses immediate update, what exactly does the recovery process do for T1 and T2? How would this differ if the system used deferred update?

NoSQL Databases vs SQL Databases

Relational databases (SQL databases) and NoSQL databases are designed with different goals in mind. The rise of NoSQL ("Not Only SQL") databases was driven by limitations of traditional RDBMS in certain scenarios, especially at **web scale**. Some key differences in properties:

- **Schema Flexibility:** SQL databases have a fixed schema defined by a data definition language (tables with specified columns and types). NoSQL databases often have a **looser or dynamic schema** – e.g., each document in a MongoDB collection can have a different structure. This makes NoSQL attractive for applications where the data model evolves or is semi-structured.

- **Scalability:** Relational DBs were traditionally scaled **vertically** (bigger single server), and while many can be distributed, it is complex. NoSQL systems are typically designed to **scale out horizontally** across commodity servers, partitioning (sharding) data across nodes and replicating for fault tolerance. This makes them better suited for very large datasets or very high throughput on commodity hardware.

- **Data Models:** RDBMS use tables (rows and columns) and support joins between them. NoSQL offers a variety of data models (key-value, document, column-family, graph) to better fit certain data shapes (discussed in the next section).

- **Query Capabilities:** SQL databases have a powerful, standardized query language (SQL) that supports complex joins, aggregations, etc. NoSQL systems often sacrifice some of this flexibility for speed or partition tolerance. Many NoSQL databases use simpler query APIs or no join support to maintain performance or simplicity. However, some NoSQL solutions and NewSQL systems are adding back SQL-like queries or limited joins.

- **Consistency and Transactions:** RDBMSs emphasize **ACID** – strong consistency and fully atomic transactions. NoSQL databases often relax ACID guarantees for the sake of performance or availability. Many NoSQL systems choose eventual consistency (where data updates propagate to replicas over time, and reads might temporarily see stale data) over strict consistency. That said, some NoSQL databases do support certain ACID properties (for example, MongoDB supports atomic operations on single documents, and newer versions even multi-document transactions). Overall, the philosophy is often **"ACID when needed, BASE by default."**

ACID vs. BASE

In NoSQL discussions, **BASE** is an acronym used as a counterpoint to ACID:

- **Basically Available:** The system is designed to be operational basically all the time. This implies some partial failures or latency might occur, but the system as a whole remains available (it doesn't shut down everything on a fault).

- **Soft state:** The state of the system may change over time, even without input, due to asynchronous updates. There isn't a hard guarantee that repeated reads will return the same value if no new writes were issued (contrast with ACID isolation).

- **Eventual Consistency:** The system will eventually converge to a consistent state if no new updates occur. Temporary inconsistency is allowed; replicas might have different versions of data for a time, but they will sync up in the long run.

In other words, ACID is pessimistic (consistency first, system may deny some operations to preserve it), while BASE is optimistic (availability first, consistency is achieved later). For example, consider a distributed shopping cart service. An ACID approach might lock a user's cart across multiple nodes to ensure all updates are immediately visible everywhere (at the cost of availability if the network is partitioned). A BASE approach might allow the cart to be slightly inconsistent (maybe two nodes briefly think the cart has a different item count) but ensure that within a second or two after updates stop, all nodes agree on the cart contents.

Modern systems sometimes blend these approaches (tunable consistency, or transactions on subsets of data). The choice of ACID vs BASE often comes down to the **CAP theorem** and application needs.

NoSQL Data Models

Unlike one-size-fits-all relational tables, NoSQL encompasses multiple data modeling approaches, each suited to different use cases:

- **Key-Value Stores:** The simplest NoSQL model, essentially a huge hash table or dictionary that associates **keys** with **values**. An application can put and get values by key. The database doesn't impose a schema on the value – it could be a text string, JSON, BLOB, etc. These systems excel in speed and scalability for simple data access patterns (e.g., caching, user session data). *Use cases:* Caching systems, simple user profile storage, preference settings. *Examples:* Redis, Memcached, Amazon DynamoDB (has key-value behaviors), Riak.

- **Document Stores:** These also use a key to identify data, but the value is a structured **document**, often stored in JSON or BSON format. The document can have many fields and nested sub-documents. Document databases understand the document structure, allowing query by fields inside the document (e.g., find all documents where `user.name = "John"`). They are great for semi-structured data that doesn't fit neatly into tables and for situations where data for an entity (which might span multiple tables in an RDBMS) is more naturally stored as one document. *Use cases:* Content management, catalogs, user profiles, event logging. *Examples:* MongoDB, CouchDB, Firebase Firestore. (In a sense, HTML or JSON documents stored in these DBs are self-describing – each can have its own schema.)

- **Column Family Stores (Wide-Column Stores):** These organize data into rows and columns, but unlike RDBMS, each row can have a variable number of columns and column groupings called **column families**. It's somewhat like a two-level map: Row key -> ColumnFamily -> (ColumnName -> Value). They are optimized for querying large datasets where you often fetch a subset of columns. They arose from Google's Bigtable design. *Use cases:* Big data analytics, time-series data, large distributed data stores where denormalization is preferred. *Examples:* Apache Cassandra, Apache HBase, ScyllaDB. In these systems, data is often modeled as one big table per query type, heavily denormalized (since joins are not native), and queries retrieve one row's multiple columns efficiently.

- **Graph Databases:** These are designed for data where relationships are first-class citizens. The data model is a graph: entities are **nodes** and relationships are **edges** connecting nodes. Each node or edge can have properties. Graph DBs excel at queries about connections, such as finding shortest paths, traversing friends-of-friends, or exploring networks. *Use cases:* Social networks, recommendation systems, fraud detection (e.g., finding rings of transactions), network topology, knowledge graphs. *Examples:* Neo4j, Amazon Neptune, JanusGraph. Instead of joins, you navigate through edges; the databases are optimized for traversals.

Each NoSQL type makes different trade-offs. For example, key-value and document stores often sacrifice complex query capability (or require adding secondary indexes) in exchange for speed and partition tolerance. Graph databases sacrifice horizontal partitioning ease for complex relationship querying power. The choice depends on the nature of the data and access patterns.

NoSQL database types overview – key characteristics and examples of key-value, document, graph, and column-oriented databases.

(Image: Summary of NoSQL data models with example use cases and popular systems.)

Practice Questions: NoSQL Data Models

- Match each of the following use cases to a NoSQL data model:

    1. A social network that needs to find mutual friends and suggest connections.

    2. A caching system for session data where each session is just a blob of JSON identified by a session ID.

    3. A huge table of sensor readings where each sensor logs a timestamp and various measurements, and we query time ranges often.

    4. A content management system storing articles that each have varying fields and embedded media.

- For the four primary NoSQL types (Key-value, Document, Column-family, Graph), what are the primary advantages of each?

- If you were designing a recommendation engine that stores users, items, and relationships like "User A likes Item X", which NoSQL database type might you choose and why?

- How do document databases differ from key-value stores? What added functionality do they typically provide?

CAP Theorem and Distributed System Trade-offs

The **CAP theorem** is a fundamental principle in distributed database design, particularly relevant to many NoSQL systems. It states that a distributed data store **cannot simultaneously guarantee all three** of the following: **Consistency**, **Availability**, and **Partition Tolerance**

mwhittaker.github.io
. In the presence of a network partition, a choice must be made between consistency and availability
en.wikipedia.org

en.wikipedia.org
:

- **Consistency (C):** Every read receives the most recent write or an error
  en.wikipedia.org
  . This is similar to the database concept of linearizability – all nodes present a unified, up-to-date view of data. (Note: Consistency in CAP is defined differently than the *C* in ACID
  en.wikipedia.org
  ; CAP consistency is about replicas agreeing on the same latest data.)

- **Availability (A):** Every request (to a non-failing node) results in *some* response – it may return stale data, but the node will not refuse to respond due to data being unavailable
  en.wikipedia.org
  . The system remains operational and responsive on all non-failed nodes.

- **Partition Tolerance (P):** The system continues to operate despite network partitions
  en.wikipedia.org
  . A network partition means communication is lost between some subset of nodes – effectively the network splits into two or more sets that cannot talk to each other. Partition tolerance is essential in any distributed system that can't guarantee a perfect network; it means the system can work even if messages are dropped or delayed.

In a normal, non-partitioned situation, a system can be both consistent and available. The CAP theorem really bites during a **partition**: when nodes cannot all communicate, you have to pick: do you choose to remain consistent (C) or remain available (A)?

- **CP (Consistent + Partition-tolerant):** The system sacrifices availability. In a partition, it will refuse some requests or return errors rather than serve stale data. Many distributed SQL databases or strongly consistent systems (like MongoDB in a single-primary mode, or Google's Spanner) lean CP – if you can't contact a majority of replicas, you stop accepting writes to remain consistent.

- **AP (Available + Partition-tolerant):** The system sacrifices consistency. In a partition, each side continues operating, possibly diverging. All requests are served, but some reads might not see the latest writes (they see what's available on that partition). Eventually, when the partition heals, the differences must be reconciled (eventual consistency). DynamoDB and Cassandra are often cited as AP systems – they will accept writes on each partition and sync up later.

- **CA (Consistent + Available) but not partition-tolerant:** This is essentially a single-node system or tightly coupled cluster that fails if partitioned. Traditional RDBMS on a single server is CA (always consistent and available as long as the server is up, but if the server is isolated from clients, it's just downtime, as it cannot tolerate a partition). CA can also be achieved in a cluster that uses failover in such a way that if the cluster is split, one side is completely shut down (so not really available to that side of the partition).

In reality, network partitions *will* happen in any large distributed system (or at least, you cannot guarantee they won't), so **partition tolerance is not optional** – any distributed system has to handle P to some degree. Thus, the practical trade-off is between C and A when P occurs

en.wikipedia.org

. Eric Brewer (who formulated CAP) clarified that CAP doesn't mean choose *one* of the three to drop forever; it means in a partition you must drop either consistency or availability. Outside of a partition, you can have both.

Example: Imagine a distributed database with replicas in New York and Los Angeles. Suddenly, network connectivity between the coasts fails (partition). If the system is designed for **consistency (CP)**, one replica (or both) might stop allowing updates or reads that can't be validated as up-to-date. Maybe the LA node refuses writes because it can't confirm with NY, thus some users in LA get errors (not available). If it's designed for **availability (AP)**, both NY and LA continue taking writes independently. A user in NY might not see a write that a user in LA made a moment ago (inconsistency), but both can continue working. When the network is restored, the system will reconcile any conflicting updates (this might be complex, potentially using strategies like "last write wins" or merging records).

It's important to note CAP presents a simplified model. Real-world systems often aim for a balance: **tunable consistency** (letting the client choose how many replicas must confirm a read/write, trading off consistency vs availability), or offering both strong and eventual consistency for different operations. Also, CAP is about *system-level guarantees*; within a single node, a database might still use transactions (ACID) for operations on that node, but across partitions it might relax consistency.

Visualization of the CAP theorem's trade-offs. In a network partition, a distributed system can either maintain consistency (all nodes show the same data) or availability (every request gets a response), but not both.

(Image: Venn/Euler diagram illustrating CAP options: CA (no partition tolerance), CP, and AP combinations.)

Practice Questions: CAP Theorem

- Define Consistency, Availability, and Partition Tolerance in the context of CAP (contrast CAP's definition of consistency with the database transactional consistency).

- If a database claims to be AP (Available and Partition-tolerant), what behavior should you expect during a network partition? What might clients experience in terms of data staleness?

- Give a real-world scenario (not necessarily computing) that is analogous to CAP trade-offs (for example, consider two people trying to keep a story consistent over a patchy phone line – do they wait to speak until the line is clear, or do they keep talking regardless of breaks?).

- Why is the CA combination not feasible in a distributed system that can have partitions? What is an example of a CA system (hint: think non-distributed or single point of failure)?

- Many NoSQL databases default to eventual consistency. How does this choice relate to the CAP theorem? Which letter of CAP are they prioritizing and which are they relaxing?

MongoDB: Document Database Overview

MongoDB is a popular NoSQL database that falls into the **document store** category. Rather than tables, it stores data in flexible, JSON-like documents. Key concepts in MongoDB's data organization:

- A MongoDB **instance** (server) can host multiple **databases** (analogous to separate databases in an RDBMS).

- Each database holds a set of **collections** (similar to tables in SQL, but without a fixed schema).

- A **collection** contains multiple **documents** (analogous to rows, but each document is a self-contained data structure, typically represented as BSON – binary JSON).

- A **document** is essentially a JSON object: a set of field-value pairs (and values can include numbers, strings, arrays, nested objects, etc.). Each document in a collection can have its own unique set of fields, though common structure is typical.

For example, a SQL table **Customers(id, name, email, age)** might translate to a MongoDB collection where each document looks like:

json
CopyEdit
```
{
   "_id": ObjectId("..."),   // a unique identifier automatically
provided if not given
   "name": "Alice",
   "email": "alice@example.com",
   "age": 30,
   "address": { "city": "Boston", "zip": "02115" }  // a nested
subdocument
}
```

Another document in the same collection could have extra fields or omit some fields:

json
CopyEdit
```
{
   "_id": ObjectId("..."),
   "name": "Bob",
   "email": "bob@example.com",
   "hobbies": ["guitar", "hiking"]
}
```

This flexibility (no enforced schema) is a major feature of MongoDB. It means no costly schema migrations for adding new fields; the application can simply start using new fields.

MongoDB has many features that make it comparable to relational DBs in capability:

- It supports **indexes** on document fields to speed up queries (B-tree indexes, etc.).

- It supports a rich **query language** (not SQL, but you can query by field conditions, do aggregations, etc. using JSON-based query syntax).

- It ensures **atomicity at the document level** – any update to a single document (which may contain many nested fields) is atomic. This is important because MongoDB initially did not support multi-document transactions; the data model encourages putting related data in one document so that updates to that data can be done atomically on one document.

- It supports **replication** through replica sets. MongoDB can be configured in a primary-secondary replication mode (one primary node receives writes, secondaries replicate from it). This provides high availability (if the primary fails, a secondary is automatically elected as the new primary).

- It supports **sharding** (horizontal partitioning) for scalability. A collection can be distributed across shards (different servers), each holding a subset of the documents, based on a shard key.

- In newer versions (since MongoDB 4.0), it even supports multi-document **ACID transactions** (so you can make a transaction that spans multiple documents and even multiple collections, with commit/rollback).

Terminology mapping from RDBMS to MongoDB is useful to recall:

- **Database** in SQL is **Database** in MongoDB (both use the term).

- **Table** in SQL corresponds to a **Collection** in MongoDB.

- **Row** in a table corresponds to a **Document** in a collection.

- **Column** in SQL corresponds to a **Field** in MongoDB.

- **Primary Key** in SQL is usually `_id` in Mongo (each document has a unique `_id` field, automatically indexed).

- **Joins** in SQL – MongoDB doesn't use joins as a query mechanism in the same way. Instead, it encourages data to be nested (one-to-many relationships can be handled by embedding an array of subdocuments). For relationships that aren't embedded, references can be used (store another document's ID and manually resolve, or use the `$lookup` stage in aggregation which is akin to a join). Recent MongoDB versions have improved join-like capabilities via the aggregation framework.

- **Transactions** – historically SQL supports complex transactions, MongoDB originally limited transactions to single-document operations but now does multi-document transactions too (with some limitations, e.g., all docs must be on the same shard if sharded).

Use Cases for MongoDB: It's often used when you need schema flexibility or JSON storage. For example, storing user profiles or content objects that have varying attributes, or for rapid prototyping when your schema is evolving. Also, since it can store hierarchical data easily, it's useful for things like storing comments (tree of replies) or product catalogs with nested specifications.

MongoDB Example – Comparing SQL and Mongo Operations: Suppose we want to store a user and their email addresses.

- In a SQL schema, you might have a `Users` table and a separate `Emails` table (for multiple emails per user) linked by user_id. To insert a user with two emails, you'd do an INSERT into `Users` and two INSERTs into `Emails`, possibly wrapped in a transaction.

- In MongoDB, you could just insert one document into the `contacts` (or users) collection with an array of emails as a field. That single insert captures the user and their emails in one document.

Practice Questions: MongoDB

- In MongoDB, you have a collection `Books`. Give an example of a document for a book in JSON format. Now, if we wanted to add a new field "publisher" to some books, how is this handled differently in MongoDB vs a relational database?

- What is a MongoDB collection and how does it differ from a SQL table in terms of schema enforcement?

- Explain how MongoDB achieves high availability through replica sets. What happens when a primary node in a MongoDB replica set goes down?

- Describe one advantage and one disadvantage of embedding data in a single MongoDB document (for example, an order document that contains an array of all item sub-documents for that order) versus splitting the data into multiple collections.

- If you needed to perform a transaction that modified two different documents (say, two separate customer records) in MongoDB, what feature would you use in modern MongoDB to ensure atomicity?

- **Challenge:** How might you model a one-to-many relationship (like blog posts and comments) in MongoDB? Give two approaches (embedding vs referencing) and discuss when you'd use each.

Querying in MongoDB with Compass (JSON Queries)

MongoDB Compass is a graphical user interface (GUI) for MongoDB that allows querying your data visually. Whether using Compass or the MongoDB shell/driver, MongoDB queries are expressed in JSON (BSON) format, quite different from SQL. A basic MongoDB find query consists of a **filter document** (criteria) often with special **operators**.

Basic Query Structure: A query is a JSON object `{ ... }` that specifies field conditions. For example:

json
CopyEdit
```
{ "name": "Joe" }
```

This query finds all documents where the field `name` is exactly "Joe". It's equivalent to the SQL: `SELECT * FROM collection WHERE name = "Joe"`.

If you provide an empty filter `{ }`, that means "no conditions" – i.e., select all documents.

For more complex conditions, MongoDB uses **query operators** prefixed with `$`. The general structure for using an operator is:

json
CopyEdit
```
{ "<field>": { "<operator>": <value> } }
```

For example, to find documents where `status` is either "A" or "D":

json
CopyEdit
```
{ "status": { "$in": ["A", "D"] } }
```

Here `$in` is an operator meaning the field's value should be *in* the given array. This is analogous to SQL `WHERE status IN ("A","D")`. There are many such operators: `$gt` (greater than), `$lt` (less than), `$ne` (not equal), `$exists` (field exists), etc.

Combining Conditions – AND/OR: In MongoDB:

AND: If you just list multiple field conditions in the same JSON object, Mongo assumes an AND. E.g.:

json
CopyEdit
```
{ "status": "A", "qty": { "$lt": 30 } }
```
finds documents where `status` is "A" **and** `qty` < 30. This can also be written explicitly as:

json
CopyEdit
```
{ "$and": [ { "status": "A" }, { "qty": { "$lt": 30 } } ] }
```

- which is equivalent.

OR: To combine with OR, use the `$or` operator with an array of conditions:

json
CopyEdit
```
{ "$or": [ { "status": "A" }, { "qty": { "$lt": 30 } } ] }
```

- This finds documents where either `status` is "A" or `qty` < 30.

You can mix these to create complex queries, similar to using parentheses in SQL. For example, find documents where status is "A" **and** (qty < 30 **or** item starts with "P"):

json
CopyEdit
```
{
  "status": "A",
  "$or": [
    { "qty": { "$lt": 30 } },
    { "item": { "$regex": "^P" } }
  ]
}
```

(This uses `$regex` operator for a pattern match on item.)

Using Compass: In MongoDB Compass, you typically select a collection and then in the “Filter” box you can type a query in this JSON format. Compass will help with syntax highlighting and sometimes suggestions. For instance, to use the earlier example, you’d type in the filter box:

bash
CopyEdit
```
{ status: "A", qty: { $lt: 30 } }
```

and Compass would display the documents meeting that criteria.

Example: Suppose we have a `products` collection, and we want to find products of category "electronics" that have a quantity between 50 and 100. A MongoDB query might be:

json
CopyEdit
```
{
  "category": "electronics",
  "qty": { "$gte": 50, "$lte": 100 }
}
```

This is analogous to `WHERE category='electronics' AND qty BETWEEN 50 AND 100` in SQL. In Mongo, we used `$gte` (>=) and `$lte` (<=) on the same field; multiple conditions on the same field combine with AND as well (both must be true for `qty`).

Another example, if you wanted all documents where `type` is "book" or "magazine" and `inStock` is true:

json
CopyEdit
```
{
  "type": { "$in": ["book", "magazine"] },
  "inStock": true
}
```

In SQL: `WHERE type IN ('book','magazine') AND inStock = true`.

MongoDB’s query language also has powerful aggregation and update capabilities, but for basic find queries, mastering this JSON syntax is key. It’s intuitive once you get used to it: think of constructing the query as an object describing the conditions.

Practice Questions: MongoDB Queries

- Write a MongoDB query (in JSON object format) to find all documents in a `users` collection where the user's age is greater than 25.
- How would you express the SQL condition `WHERE score >= 90 AND score <= 95` in a MongoDB query on a field `score`?
- If you want to find all documents where `status` is "active" **or** `lastLogin` is more than 30 days ago, what would the MongoDB query look like (hint: use `$or` and a date comparison operator like `$lt` on `lastLogin` field)?
- Explain the difference between providing multiple conditions in one JSON (implicitly ANDed) vs using `$or`. Give an example where these yield different results.
- In Compass, what does an empty filter `{ }` do?
- **Challenge:** There is a `orders` collection. Each order document has a field `items` which is an array of strings (product names). How would you find all orders that include the item "bananas" in the items array? (Hint: Mongo query operators can match array contents, e.g., field: "value" works on array fields too.)

Section 1: Transaction and ACID

Q1: Define “transaction” in the context of databases. Why is it called a logical unit of work?

Answer:

- A **transaction** is a group of operations (reads and writes) performed on the database that form a single, indivisible unit of work. It is the smallest sequence of operations that must be executed in its entirety to ensure data integrity.
- It is called a **logical unit of work** because from the user’s or application’s perspective, the grouped operations accomplish a single business function (e.g. transferring money from one account to another) and must therefore either all succeed or all fail together.

Q2: Explain each of the ACID properties in your own words and give a real-world analogy or example for each.

Answer:

1. **Atomicity**: “All or none.” Either every operation in a transaction succeeds, or none do.
    - *Real-world analogy:* When you buy an item online, the order must deduct your balance and add the order to the seller’s account. If one part fails (out-of-stock, credit card error), the entire order is canceled.
2. **Consistency**: The transaction preserves all database rules and constraints, thus moving the database from one valid state to another valid state.
    - *Analogy:* If a rule says a store can’t sell more products than it has in inventory, any transaction that tries must fail or adjust properly so the rule still holds after the transaction.
3. **Isolation**: Concurrent transactions don’t interfere with each other; each executes as if it were the only one.
    - *Analogy:* If two people are editing different sections of a shared document at the same time, each sees only their own changes until each saves, preventing partial or inconsistent views.
4. **Durability**: Once a transaction commits, its changes are permanent, surviving power outages, crashes, etc.

    - *Analogy:* After you confirm a bank transfer, it's recorded so that even if the system restarts, the new balance remains.

Q3: What happens if a transaction fails midway and why is atomicity crucial in that case?

Answer:

- If a transaction fails midway, any partial changes must be **rolled back** so that the database returns to the state before the transaction began.
- **Atomicity** is crucial here because it guarantees that partial, unfinished updates do not remain in the system; it's either as if the transaction never started or as if it completed fully.

Q4: A system crashes right after a transaction commits. Which ACID property ensures the changes are not lost, and how might the DBMS implement this?

Answer:

- The **Durability** property ensures changes are not lost after a commit.
- The DBMS typically implements this using **write-ahead logging**: once a transaction commits, all changes are recorded in a stable log on disk. If the system crashes, the log is used during recovery to "redo" any committed transactions that didn't fully persist.

Section 2: Concurrency Control and Serializability

Q1: What is the difference between a serial schedule and a serializable schedule? Why do we allow nonserial schedules at all?

Answer:

- A **serial schedule** runs transactions one by one, with no interleaving, so it trivially avoids concurrency issues.

- A **serializable schedule** interleaves operations from different transactions but ensures the final outcome is the same as some serial schedule.

- We allow nonserial schedules to improve performance (more concurrency), as purely serial execution underutilizes the system when transactions wait idly for others to finish.

Q2: Give an example of a lost update problem. How does concurrency control prevent it?

Answer:

- *Example:* T1 withdraws $10 from an account balance of $100, T2 deposits $100 into the same account. If T1 and T2 read $100 simultaneously, T1 writes back $90, and T2 writes back $200, one update is lost. The correct final balance should be $190 but ends up $200 or $90.

- Concurrency control prevents this by using locks or other protocols to ensure these two write operations on the same data item do not overwrite each other blindly; one transaction must complete (or update the item) before the other modifies the same data item.

Q3: Define conflict serializability. How would you determine if a given interleaved schedule is conflict-serializable (what tool or test is used)?

Answer:

- **Conflict serializability** is a stricter condition in which two schedules are equivalent if every pair of conflicting operations (i.e., operations on the same data item where at least one is a write) appear in the same order.

- We use a **precedence graph** (serialization graph). We create one node for each transaction, and draw edges if operations conflict. If the graph has no cycles, the schedule is conflict-serializable.

Q4: Explain what a recoverable schedule is. Why is the schedule where T2 reads from T1 and commits before T1 commits problematic?

Answer:

- A **recoverable schedule** ensures that if T2 reads a value T1 wrote, T2 cannot commit until T1 commits. Otherwise, if T1 later aborts, T2 has read invalid data (dirty read).

- The schedule is problematic because T2 depends on T1's uncommitted data. If T1 aborts, T2's commit includes changes based on a never-committed update. This could corrupt the database or produce inconsistencies.

Q5: What is a cascading abort? How do cascadeless schedules eliminate cascading aborts?

Answer:

- A **cascading abort** occurs when a transaction abort causes another transaction to abort in turn because it used data from the first transaction. This can lead to multiple rollbacks.

- **Cascadeless schedules** ensure no transaction reads uncommitted data from another transaction. Hence, if T1 aborts, no other transaction depends on T1's uncommitted data, so no chain reaction of aborts is needed.

Q6 (Challenge): Draw a small precedence graph for two transactions that conflict and show how a cycle indicates a non-serializable schedule.

Answer (conceptual):

1. Suppose T1: R(X), W(X). T2: R(X), W(X).

2. An interleaving might be: T1: R(X) → T2: R(X) → T2: W(X) → T1: W(X).

3. For the graph, we have an edge T1 → T2 (T2 does a W(X) after T1's R(X)) and T2 → T1 (T1 does W(X) after T2's R(X)). This forms a cycle T1 → T2 → T1, so it's not conflict-serializable.

Section 3: Locking and Deadlocks

Q1: Explain how two-phase locking ensures serializability. What are the two phases and why is the protocol important?

Answer:

- **Two-phase locking (2PL)** ensures transactions acquire and release locks in a way that prevents cycles of conflicts. It has two phases:

    1. **Growing phase**: a transaction can acquire locks but not release any.
    2. **Shrinking phase**: once it releases a lock, it cannot acquire new locks.

- This approach prevents cyclical waiting among transactions and guarantees conflict-serializability. Once a transaction starts releasing locks, it cannot lock additional items, thus avoiding certain concurrency anomalies.

Q2: What is the difference between a shared lock and an exclusive lock? Give an example scenario for each.

Answer:

- A **shared lock (S)** allows multiple concurrent transactions to read the same data item. For example, multiple customers can read a product's price simultaneously.
- An **exclusive lock (X)** gives one transaction exclusive rights to read and write that item. For instance, if a transaction needs to update a bank account balance, it must acquire an exclusive lock on that balance to prevent others from even reading an uncommitted new balance.

Q3: Consider a database where one transaction is updating a set of rows in a table and another is reading different rows of the same table. How could lock granularity affect their concurrency?

Answer:

- If the DBMS only supports coarse **table-level** locks, the updating transaction will lock the entire table, blocking the reading transaction even for rows not updated. This reduces concurrency unnecessarily.
- If the DBMS supports **row-level** locks, the writer locks only the specific rows it's updating, so the reader can still read other rows in the same table, improving concurrency.

Q4: Describe a deadlock scenario involving two transactions and two data items. How might the DBMS detect this situation?

Answer:

- Example deadlock:

    - T1 locks item A, T2 locks item B.
    - T1 tries to lock B, but B is locked by T2. T1 waits.
    - T2 tries to lock A, but A is locked by T1. T2 waits.
    - Both wait forever.
- The DBMS can **detect** this by building a **waits-for graph**. If there is a cycle in that graph (T1 → T2 and T2 → T1), a deadlock is detected. One transaction is chosen as victim and aborted to break the cycle.

Q5: What is strict two-phase locking and how does it relate to recoverability?

Answer:

- **Strict 2PL** is a variant of 2PL where transactions do not release their **exclusive** locks until **after** commit (or abort). This means no other transaction can read or write data modified by T until T commits.
- It ensures a **recoverable** schedule because no other transaction sees uncommitted data. So if T aborts, there's no need to cascade a rollback.

Q6: How do *Wait-Die* and *Wound-Wait* deadlock prevention strategies decide which transaction to abort?

Answer:

- Both use **timestamps**:
    - **Wait-Die**: If an older transaction needs a lock held by a younger, it waits. If a younger needs a lock held by an older, it aborts ("dies").
    - **Wound-Wait**: If an older transaction needs a lock held by a younger, it "wounds" (forces an abort of) the younger. If a younger needs a lock held by an older, it waits.
- These protocols systematically avoid cycles by always favoring the older or younger transaction consistently.

Section 4: Timestamps

Q1: How do timestamps differ from locks in managing concurrent transactions? What potential advantage do they have regarding deadlocks?

Answer:

- **Timestamp protocols** do not use locking or waiting; instead, each transaction gets a unique timestamp and the system checks for conflicts based on the transaction's timestamps. If a conflict arises, a transaction may be aborted (rolled back) rather than waiting.
- Because they do not wait, **deadlocks cannot occur**. There's no situation where T1 is waiting for T2 and T2 is waiting for T1—any conflict is resolved by aborting one transaction immediately.

Q2: Transaction T1 has timestamp 5 and transaction T2 has timestamp 10. Data item X currently has a write-timestamp of 8 and read-timestamp of 9. If T1 attempts to read X, what will happen? If T1 attempts to write X, what will happen?

Answer:

- **Reading X**: T1's timestamp is 5, which is less than the write-timestamp(8). This means X was most recently written by a newer transaction. According to the basic timestamp ordering rule, T1 is too old to read this updated value (the value changed after T1 "began"). Typically, T1 would be **aborted** and restarted with a newer timestamp.
- **Writing X**: If T1 tries to write X, the system sees T1's ts=5 < read-timestamp(9) or write-timestamp(8). Either condition triggers an **abort** of T1 because T1 is older than the last read or last write. T1 writing would violate the transaction order.

Q3: What does Thomas's Write Rule change about the basic timestamp ordering protocol? Why might a database choose to use Thomas's rule?

Answer:

- Thomas's Write Rule says that if a transaction T is trying to write an item X, but T's timestamp is *less* than X's **write-timestamp**, T's write is considered **obsolete** and simply discarded rather than aborting T.
- This avoids unnecessary aborts if T's update is older than a more recent update. The database might choose this to reduce rollbacks in situations where overwritten updates

are worthless anyway.

Q4: In timestamp ordering, why is it necessary to assign a new timestamp to a restarted transaction after it aborts? What could go wrong if we reused the old timestamp?

Answer:

- We give a new, larger timestamp so the restarted transaction is recognized as "newer" than any conflicting transactions.
- If we reused the old timestamp, the same conflict might occur again, potentially causing an infinite loop of aborts. The system would keep seeing the transaction as old.

Q5 (Challenge): Consider a scenario where T1 (ts=1) writes X, then T2 (ts=2) writes X, then T1 tries to write X again. Describe how basic timestamp ordering handles this vs how it would be handled with Thomas's Write Rule.

Answer (summary):

- **Basic timestamp ordering** sees T1's second write as older than the write-timestamp of X (which is now 2 from T2's write), so T1 must be aborted.
- **Thomas's Write Rule**: T1's second write is obsolete, so we ignore that write but allow T1 to continue, thus no abort is necessary. The final value of X is T2's write.

Section 5: Recovery

Q1: List three types of failures that a database must be able to recover from. How might each of these failures affect transactions in progress?

Answer:

1. **System crash** (hardware/software failure): Transactions in progress lose the contents of main memory buffers. Partial changes might be undone or redone.
2. **Media failure** (e.g., disk head crash): Parts of the database might be destroyed. Must restore from backups and redo committed transactions from logs.

3. **Application error** (bugs, or user abort): Uncommitted transaction is aborted and rolled back, removing partial updates.

Q2: What is the purpose of the log in database recovery? Why must the log be written to stable storage *before* the actual data is written?

Answer:

- The **log** records every change (with before- and after-images) and critical events (like transaction start, commit).
- It must be on stable storage first (**write-ahead logging**) so that if a crash occurs, the system can use the log to **undo** or **redo** any changes. If data were written without logging, we would lack the information to recover partially applied updates.

Q3: Explain the difference between deferred update and immediate update recovery techniques. What does each one require in terms of undo or redo?

Answer:

- **Deferred update**: Actual database updates occur only at commit time. Before that, changes are recorded only in a log or in-memory. If a transaction aborts, no changes are in the database to undo. **On recovery:** redo all committed transactions. No undo needed.
- **Immediate update**: Data can be updated on disk at any time, even before commit. On recovery, we must **undo** changes of uncommitted (aborted) transactions and **redo** changes of committed ones.

Q4: Describe what a checkpoint is and how it helps minimize recovery time.

Answer:

- A **checkpoint** is a point when the DBMS forces all in-memory modified pages to disk and logs a checkpoint record.
- It speeds up recovery because instead of reprocessing the entire log from the start, recovery can begin from the last checkpoint, ignoring older log records since those changes are already safely persisted.

Q5: In the event of a crash, how does the recovery manager determine which transactions to undo and which to redo?

Answer:

- In immediate-update logging systems, the **analysis** pass identifies transactions active at crash time (to be undone) or committed (possibly need redo).
- **Undo** all updates by transactions that never reached commit.
- **Redo** all updates by transactions that committed but whose writes might not have fully reached disk.

Q6 (Challenge): Consider a scenario with two transactions T1 and T2 where T1 commits before the crash and T2 does not. If the system uses immediate update, what exactly does the recovery process do for T1 and T2? How would this differ if the system used deferred update?

Answer:

- **Immediate update**: On recovery, T1's changes must be **redone** (because T1 committed but might not have fully written to disk). T2's changes must be **undone** (because T2 didn't commit).
- **Deferred update**: T1's changes might not yet have been applied to the database at commit, so we must redo T1's changes from the log. T2's changes do not appear in the database at all since it didn't commit, so no undo is needed in the actual database (though the log entries for T2 can simply be ignored).

Section 6: NoSQL Databases vs. SQL Databases

Q1: Match each of the following use cases to a NoSQL data model:

1. A social network needing to find mutual friends
2. A caching system for session data
3. A huge table of sensor readings for big data analytics
4. A content management system storing articles with varying fields

Answer:

1. **Graph** (finding mutual friends, relationships).
2. **Key-value** (session data).
3. **Column-family** (time-series sensor data).
4. **Document** (storing articles with different structures).

Q2: For the four primary NoSQL types (Key-value, Document, Column-family, Graph), what are the primary advantages of each?

Answer (brief):

- **Key-value**: Simplicity, high performance, easy to scale horizontally.
- **Document**: Flexible schema, good for semi-structured data, can query on fields in documents.
- **Column-family**: Efficient for large distributed datasets, wide tables, good for analytical queries over certain columns.
- **Graph**: First-class representation of relationships, great for connected data and graph traversals.

Q3: If you were designing a recommendation engine that stores users, items, and relationships like "User A likes Item X," which NoSQL database type might you choose and why?

Answer:

- Typically **Graph** database is best for relationship-intensive queries like "friends of friends" or "items liked by users similar to me." Graph DBs are optimized for queries that traverse nodes and edges quickly.

Q4: How do document databases differ from key-value stores? What added functionality do they typically provide?

Answer:

- In **key-value stores**, the value is opaque to the database (no inherent structure). They are basically distributed dictionaries.
- **Document databases** also store items keyed by an ID, but the value is a structured JSON/BSON document with fields. They allow querying on nested fields, indexing specific fields, partial updates, and more complex queries.

Section 7: CAP Theorem

Q1: Define Consistency, Availability, and Partition Tolerance in the context of CAP (contrast CAP's definition of consistency with the database transactional consistency).

Answer:

- **Consistency (CAP)**: All nodes see the same data at the same time; each read reflects the latest write. (Slightly different from ACID "consistency," which is about ensuring valid states under constraints.)
- **Availability**: The system continues to operate and respond to requests, even if parts of it have failed.
- **Partition Tolerance**: The system still operates despite communication failures that split the network into disconnected parts.

Q2: If a database claims to be AP (Available and Partition-tolerant), what behavior should you expect during a network partition? What might clients experience in terms of data staleness?

Answer:

- You should expect the system to remain available even if the network is split, meaning both partitions can handle reads and writes.
- Clients might experience **eventual consistency**—some reads could see stale data until the system reconciles updates after the partition is resolved.

Q3: Give a real-world scenario (not necessarily computing) that is analogous to CAP trade-offs.

Answer:

- *Analogy:* Two co-authors editing a paper from different places with a patchy internet connection. If they want perfect real-time synchronization (Consistency) whenever the line goes down (Partition), one might block the other from editing (not Available). Or they can both keep editing offline (Availability) but risk merging conflicts later (lack of immediate Consistency).

Q4: Why is the CA combination not feasible in a distributed system that can have partitions? What is an example of a CA system?

Answer:

- Because if a **Partition** occurs, you can't maintain both strong Consistency and 100% Availability across disconnected nodes. You must sacrifice one.
- A **single-node** system is basically CA (consistent and available) but not partition tolerant. If the node goes offline, it's just unavailable. Many relational databases on a single machine are "CA" as long as no partition occurs.

Q5: Many NoSQL databases default to eventual consistency. How does this choice relate to the CAP theorem? Which letter of CAP are they prioritizing and which are they relaxing?

Answer:

- They are prioritizing **Availability** (and Partition Tolerance) over strong Consistency. The system remains up and running even if nodes can't synchronize immediately. Over time, replicas converge to a consistent state.

Section 8: MongoDB

Q1: In MongoDB, you have a collection `Books`. Give an example of a document for a book in JSON format. Now, if we wanted to add a new field "publisher" to some books, how is this handled differently in MongoDB vs a relational database?

Answer:

Example document:

json
CopyEdit
```
{
  "_id": ObjectId("..."),
  "title": "Learn NoSQL",
  "author": "J. Smith",
  "year": 2022,
  "genres": ["technical", "databases"]
}
```

-
- In MongoDB, adding a field `"publisher"` is simply adding it to the document in an update. No schema migration is needed. In a relational database, you'd typically **ALTER TABLE** to add a new column, which can be more restrictive and expensive.

Q2: What is a MongoDB collection and how does it differ from a SQL table in terms of schema enforcement?

Answer:

- A **collection** is a grouping of MongoDB documents, analogous to a table in RDBMS.
- A collection does **not** enforce a rigid schema. Each document can have different sets of fields. This is more flexible compared to a SQL table that enforces fixed columns.

Q3: Explain how MongoDB achieves high availability through replica sets. What happens when a primary node in a MongoDB replica set goes down?

Answer:

- MongoDB uses **replica sets**: one node is primary (receives writes), others are secondaries. If the primary fails, the secondaries hold an election to choose a new primary. The system remains available for writes and reads with minimal downtime.

Q4: Describe one advantage and one disadvantage of embedding data in a single MongoDB document (for example, an order document that contains an array of all item sub-documents for that order) versus splitting the data into multiple collections.

Answer:

- **Advantage**: All related data is in one place, so queries for that entity require no joins. Atomic updates can be done on a single document.
- **Disadvantage**: Large documents can become unwieldy (document size limit) and updates to that document can get expensive. If certain sub-pieces are accessed frequently on their own, it can be inefficient to retrieve the entire large document.

Q5: If you needed to perform a transaction that modified two different documents (say, two separate customer records) in MongoDB, what feature would you use in modern MongoDB to ensure atomicity?

Answer:

- Use **multi-document transactions**, introduced in MongoDB 4.0. You start a transaction, update both documents, then commit or abort the transaction, ensuring atomicity across them.

Q6 (Challenge): How might you model a one-to-many relationship (like blog posts and comments) in MongoDB? Give two approaches (embedding vs referencing) and discuss when you'd use each.

Answer:

1. **Embedding**: store comments in an array within the blog post document. Good if comments are relatively small and always retrieved with the post.
2. **Referencing**: store comment documents in a separate "comments" collection, but each comment has a `post_id` referencing the blog post. Good if comments are large, or frequently accessed independently, or if the one-to-many relationship can grow large.

Section 9: MongoDB Compass Queries

Q1: Write a MongoDB query (in JSON object format) to find all documents in a `users` collection where the user's age is greater than 25.

Answer:

json

CopyEdit

```
{ "age": { "$gt": 25 } }
```

Q2: How would you express the SQL condition `WHERE score >= 90 AND score <= 95` in a MongoDB query on a field `score`?

Answer:

json

CopyEdit

```
{ "score": { "$gte": 90, "$lte": 95 } }
```

Q3: If you want to find all documents where `status` is "active" **or** `lastLogin` is more than 30 days ago, what would the MongoDB query look like (hint: use `$or` and a date comparison operator like `$lt` on `lastLogin` field)?

Answer (example):

json

CopyEdit

```
{
  "$or": [
    { "status": "active" },
    { "lastLogin": { "$lt": ISODate("2023-03-01T00:00:00Z") } }
  ]
}
```

(Adjust the date as needed; the idea is `$lt` some date threshold for 30 days ago.)

Q4: Explain the difference between providing multiple conditions in one JSON (implicitly ANDed) vs using `$or`. Give an example where these yield different results.

Answer:

- Listing multiple conditions in one object means **all** must be true (AND logic). Using `$or` means **at least one** must be true.
- Example:
  - **AND**: `{ "status": "A", "qty": { "$lt": 30 } }` → status is A and qty < 30.

  OR:

  json
  CopyEdit

  ```
  {
    "$or": [
      { "status": "A" },
      { "qty": { "$lt": 30 } }
    ]
  }
  ```

  - → either status is A or qty < 30.

Q5: In Compass, what does an empty filter `{ }` do?

Answer:

- It means “select all documents” in the collection (no filter conditions).

Q6 (Challenge): Suppose there is an `orders` collection. Each order document has a field `items` which is an array of strings (product names). How would you find all orders that include the item "bananas" in the items array?

Answer:

Simple approach:

json
CopyEdit
```
{ "items": "bananas" }
```

- If `items` is an array, matching `"items": "bananas"` returns documents where "bananas" is an element in the array. Alternatively, we could do `{ "items": { "$elemMatch": { "$eq": "bananas" } } }`, but typically the simpler approach is enough.