Data Consistency Levels

- ## Linearizable

  At this consistency level, we want to show all changes in the database until the current read request. It means **all changes which have happened in the database before the read operation will be reflected in the read query.**

  For example, suppose initially we had x = 10.

  - `update x to 13`
  - `update x to 17`
  - `read x` --> Returns 17
  - `update x to 1`
  - `read x` --> Returns 1

  To achieve this we use a **single-threaded single server**. So every read and write request will always be ordered.

  Using the above example, the first `read x` will be executed after updating the value to 17.

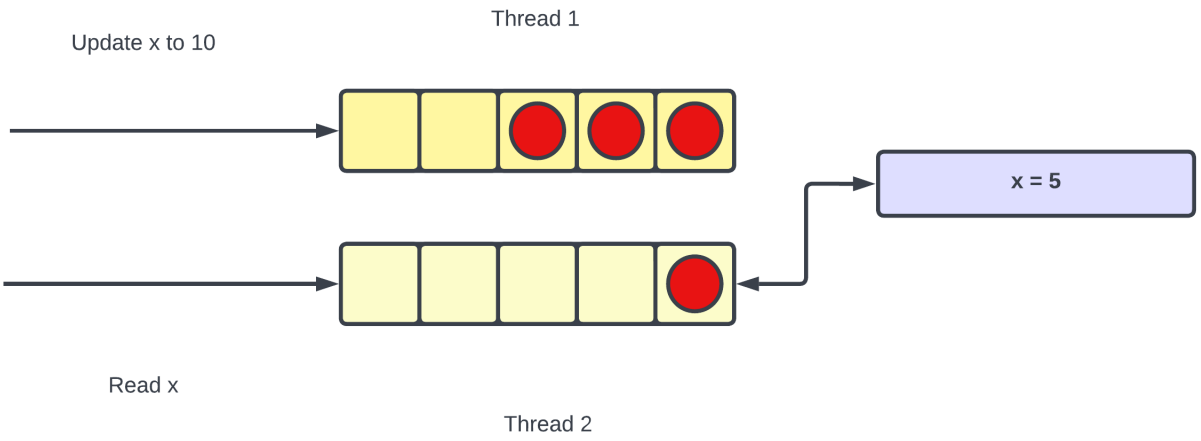
  This is useful when **systems need perfect consistency.**

- ## Eventual Consistency

  At this level, we can send stale data for a read request, but eventually, return the latest data (provided the data is not updated). **Until we are returning the stale data our system is not considered consistent but we can guarantee that after some time it will be consistent.**

  To achieve this we can process read and write requests parallelly (using multiple servers) or concurrently (using multiple threads).

  In the example below, the write request is made before the read request but the read request is processed first. So the client gets `x = 5` . But after a given time, the write request will be processed. Then all read requests will show the correct data i.e., `x=10` .



- ## Causal Consistency

  At this consistency level, if a **previous operation is related to the current operation then the previous operation has to be executed before the current operation.**

  For example

  i. `update x = 20`
  ii. `update y = 10`
  iii. `read x`
  iv. `update x = 2`
  v. `read y`

  The value of the `read x` depends only on the `update x = 20` operations. So the `update x` operation needs to be executed before the read operation. However, `update y` does not affect the value of x. So it does not matter if it is executed before or after the `read x` .

  So the 1st, 3rd and 4th operations will be executed on one server/thread and the 2nd and 5th operations will be executed on one server/thread.

  Causal consistency is better than eventual consistency because operations for the same key are processed sequentially and therefore provide better consistency.

  It is also better than linearizable consistency because it is not waiting for all the previous operations to complete and therefore provide better availability.

  Causal Consistency fails when performing aggregation operations.
  Suppose we have a table,

uid	value
1	20

---

uid	value
2	10
3	30

And we perform the following operations

i. `read sum`
ii. `update 1 to 10`
iii. `read sum`
iv. `update 1 to 5`

- If we execute the sum operations together and update operations after that then we get 60 and 60 which is wrong.
- If we execute the sum operations together and update operations before that then we get 45 and 45 which is wrong.
- If we execute the first update operation and then the sum operations we get 60 and 60 which is wrong.

We can try all possible permutations and only one way will be correct. So causal consistency does not work for aggregation operations because **causal consistency orders query based on IDs but aggregation operations use all IDs**

- ## Quorum

At this consistency level, we have multiple replicas of the database and **these replicas may or may not be in sync**. For a read query, **we get the data from all the replicas and return the most appropriate values** (Majority value, Latest updated value etc). It works on some kind of consensus in the distributed system.

Quorum is eventually consistent in most cases.

For example, we have three replicas: `x = 20` , `x = 20` , x = `20` . We update the value in the second replica to `x = 40` . Then when we make a read request second node crashes. So we will get the data from the other 2 replicas and we get `x = 20` . So it returns old data. It is eventually consistent because once the second replica is online we will get the correct result.

We can make this system strongly consistent by **specifying the minimum number of replicas we need to read the data from.** We can use the formula **R + W > N** where,

- R = **Minimum Number of replicas we need to read the data from**
- W = **Number of replicas, we write the data to**
- N = **Number of replicas**

So if we have N = 5 and W = 2, R should be greater than 3 i.e., 4. If we cannot get data from 4 replicas we send an `error` response.

It **provides fault tolerance** and depending upon the values R, W and N we can either have an eventually consistent system (**R + W<=N**) or a strongly consistent system (**R+W > N**)

Disadvantages of using quorum

- We need multiple replicas so the cost is high.
- If we have an even number of replicas then it can cause the split-brain problem.

Data Consistency Levels Tradeoffs

Level	Consistency	Effeciency
Linearizable	Highest	Lowest
Eventual Consistency	Lowest	Highest
Causal Consistency	Higer than eventual Consistency but lower than linearizable	Higer than Linearizable but lower than eventual consistency
Quorum	Configurable	Configurable

Transaction Isolation Levels

Before discussing different isolation levels, let's discuss transactions first:

Transactions can be defined as a collection of queries that perform one unit of work. They are **atomic** which means either all queries in a transaction are executed or none of it is executed.

- **BEGIN transaction** marks the starting of transactions
- **COMMIT transaction** marks the end of the transaction and persists the changes to the database
- **ROLLBACK transaction** marks the end of the transaction and undoes all the changes to the database

If two transactions are running concurrently and queries in one transaction do not affect the other transaction then the two transactions are said to be isolated from each other.

- ## Read Uncommitted

  At this isolation level, even uncommitted data can be read by concurrent transactions.

  Although it is very fast it also **leads to dirty reads**.
  Suppose we have a database where `x = 20` and we have the following transactions

T1	T2	Explanation

T1	T2	Explanation
`BEGIN`	`BEGIN`	Transactions begin
`Write x = 10`	`Read y`	value of x is updated to 10
`Read y`	`Read x`	T2 reads value of x as 10
`ROLLBACK`	`COMMIT`	T1 rollbacks and value of x becomes 20

In the second step, T1 updates x to 10. Since transactions can read uncommitted data T2 reads x as 10. But T1 rollbacks so the value of x again becomes 20. So **T2 read a value that does not exist**. This is called the dirty read phenomenon.

- ## Read Committed

At this isolation level, one transaction can only read committed data from other transactions.
Consider the same example,
Initially `x = 10`

T1	T2	Explanation
`BEGIN`	`BEGIN`	Transactions begin
`Write x = 10`	`Read y`	value of x of T1 is updated to 10
`Read y`	`Read x`	T2 reads value of x as 10
`COMMIT`	`Read y`	T1 commits
`--`	`Read x`	Now T2 reads the value of x as 10 because T1 commited

However, it leads to Non-Repeatable reads. A non-repeatable read occurs when a **transaction reads the same row twice but gets different data each time.**

Consider the following transactions (Initially x = 10)

T1	T2	Explantaions
`BEGIN`	`BEGIN`	Transaction begins
`Write x = 20`	`Read x`	T1 updates x to 20 but T2 reads the value of x as 10
`COMMIT`	`Some query`	T1 Commits

T1	T2	Explantaions
`--`	`Read x`	T2 reads the value of x as 20. So in a same transaction we get different values of x

- ## Repeatable Reads

  At this isolation level, **we make sure that when a query reads a row that row will remain unchanged while the transaction is running.**
  So we are not taking the latest committed values to get more efficiency in the repeated reads.
  Each transaction has its copy of data where they update the data. Once the transactions are committed then only the updates are persisted in the database. This is known as **Snapshot Isolation.**
  If two of the transaction concurrently change the same key to different values and we roll back the transaction it is called **Optimistic Concurrency Control**.

- ## Serializable

  This is the highest isolation level. At this level, **all transactions** are executed serially. All of the operations **are executed serially or every operation is ensured to not meddle with operations of other transactions.**
  We use this isolation level when we want to avoid **Phantom Reads**.
  A Phantom Read occurs when two identical read queries are executed and the collection of rows returned by the second query is different from the first. It is different from non-repeatable reads because in phantom reads the change in the value can be due to the insertion of a new row.
  Consider the following example,
  Suppose we have a table.

ID	Value
1	20
2	10

  Now if we execute the following transactions,

T1	T2	Explanation
`Sum`	`some query`	T1 reads 30
`Some query`	`Insert 3 with value of 30`	New value is inserted
`Some query`	`COMMIT`	T2 commits
`Sum`	`--`	T1 reads 60

Implementing Transaction Levels

Isolation Level	Implementation	Explanation
Read Uncommitted	**Single Data Entry**	We only need a single entry in the database and it is overwritten whenever there is a update operation
Read Committed	**Local Copy of Changed values**	If we are making an update to a key then the older value of key stays in the database and the newer value is kept in the local copy till the commit finally goes through.
Repeatable Read	**Versioning of Unchanges Values**	We take the values that we care about but we are not changing and keep a version of them. For every key we will store all the values that it has ever had in different transaction commits
Serializable	**Queued Locks**	We use causal ordering here. If two transactions use queries for the same key then they must be ordered. Transactions that do not have any conflict can run concurrently

For Efficiency

Read Uncommitted > Read Committed > Repeatable Read > Serializable

For Isolation

Read Uncommitted < Read Committed < Repeatable Read < Serializable

That's it for now!

You can check out more designs on our video course at InterviewReady.