Understanding Read Committed Isolation

A fundamental concept in database transactions for ensuring data consistency.

Database isolation levels are rules that determine how and when changes made by one operation become visible to others. They are crucial for preventing data corruption in systems where multiple users are reading and writing data simultaneously. **Read Committed** is one of the most common default isolation levels used in modern databases (like PostgreSQL and Oracle).

Core Guarantees:

- **No Dirty Writes:** Your transaction will not overwrite changes from another transaction that hasn't been finalized (committed) yet.
- **No Dirty Reads:** Your transaction will only read data that has been successfully finalized (committed).

✅ Topic Summary: Read Committed at a Glance

⚙️ Background: Multi-threaded Databases

- Databases process multiple reader and writer threads concurrently.
- Because operations overlap, the exact order isn't guaranteed, which can cause **race conditions**—situations where results depend on unpredictable timing.

📝 Key Term: Commit

- To **commit** means a write is confirmed and made durable in the database.
- For a transaction with multiple steps, all must succeed for the commit to happen. Partial changes are not committed and can cause inconsistencies.

⚡ Race Condition #1: Dirty Write

What is it?
Writing over uncommitted changes from another transaction.

Fix:
Use row-level locks. Only one writer can hold the lock at a time.

⚡ Race Condition #2: Dirty Read

What is it?
Reading uncommitted data from a transaction that may fail later.

Fix:
Store the old version of a row. Readers see the old, committed value until the new one is committed. This is faster than locking for reads.

📌 The Takeaway

A database using **Read Committed** prevents both **Dirty Writes** (by locking writes) and **Dirty Reads** (by versioning data for readers). It's a solid baseline for protecting data correctness while maintaining good performance. However, it doesn't protect against all race conditions.

🎥 Closing Tip: The Trade-off

Isolation levels are a trade-off. Higher isolation means more guarantees and fewer errors, but often leads to more locks, less concurrency, and slower performance.

More Examples & Analogies

1️⃣ Real-life Analogy: A Group Project

Think of editing a shared report. If you print it while your friend is still writing their part, you might submit an unfinished, incorrect version. Read Committed is the rule: **"Only print the final version after everyone says they are done!"**

2️⃣ What can go wrong without Read Committed

📍 Dirty Write

User A updates a product price to ₹500. User B updates it to ₹450 at the same time. Without locking, one update overwrites the other, and the final price is unpredictable.

📍 Dirty Read

A finance report reads uncommitted data showing a profit of ₹1M. When the transactions roll back, the real profit is only ₹500K. This causes huge confusion!

3️⃣ Simple SQL Example

Suppose you have a table `accounts` with `user_id = 1` and `balance = 10000`.

Transaction 1 (T1) - The Payer

```
BEGIN;
UPDATE accounts
SET balance = balance - 1000
WHERE user_id = 1;
-- Not committed yet!
```

Transaction 2 (T2) - The Checker

```
-- Running concurrently
SELECT balance
FROM accounts
WHERE user_id = 1;
```

- With **READ UNCOMMITTED**, T2 sees the balance as **₹9000**, even though T1 isn't done.
- With **READ COMMITTED**, T2 still sees **₹10,000** until T1 executes `COMMIT;`.

4️⃣ Key Takeaway: Isolation Level Comparison

Isolation Level	Dirty Write	Dirty Read	Non-repeatable Read	Phantom Read
Read Uncommitted	❌ Allowed	❌ Allowed	❌ Allowed	❌ Allowed
Read Committed	✅ Prevented	✅ Prevented	❌ Allowed	❌ Allowed

Read Committed stops the basic issues, but for stronger protection (like preventing non-repeatable or phantom reads), you need higher levels like **Repeatable Read** or **Serializable**.

In-Depth Example 1: Preventing Dirty Writes

Let's revisit the "dirty write" scenario with a step-by-step example. This occurs when one transaction overwrites the uncommitted changes of another, leading to lost data.

Scenario: Booking the Last Concert Ticket

Time	Alice's Transaction (T1)	Bob's Transaction (T2)
1	`UPDATE... SET purchaser = 'Alice'`	
2		`UPDATE... SET purchaser = 'Bob'`

The Problem (Without Read Committed)

Bob's update could overwrite Alice's uncommitted change. Alice's purchase would be lost, even if her transaction commits first.

The Solution (With Read Committed)

Alice's transaction locks the row. Bob's transaction is forced to wait until Alice commits, preventing the lost update.

In-Depth Example 2: Preventing Dirty Reads

A "dirty read" occurs when one transaction reads data from another, uncommitted transaction. Here's how Read Committed stops this.

Scenario: A Bank Transfer

Initial State: Savings = $500, Checking = $0.

Time	Transfer Transaction (T1)	Balance Check (T2)	Notes
1	`UPDATE Savings...`		Deducts $100 (Uncommitted)
2		`SELECT FROM Savings`	Reads Savings balance
4	`UPDATE Checking...`		**This step fails!**
5	`ROLLBACK;`		Changes are reverted.

The Problem (Without Read Committed)

The Balance Check (T2) would read the Savings balance as $400, an incorrect value since the transfer ultimately failed and was rolled back.

The Solution (With Read Committed)

T2 reads the last committed value of the Savings balance ($500). It is unaffected by T1's temporary, uncommitted changes.

Final Summary & Next Steps

As the video mentions, Read Committed is a great start, but it doesn't solve every concurrency problem. Understanding its limitations is the key to knowing when you need a stronger isolation level.

Pesky Race Conditions Still Allowed:

Non-Repeatable Read
This happens when your transaction reads the same row twice but gets different data each time, because another transaction committed an update in between your reads.

Example: You check a product price ($50), another user updates it to $45 and commits, and when you check again, you see $45. The value changed within your single transaction.

Phantom Read
This occurs when your transaction runs the same query twice and gets more rows the second time, because another transaction inserted new rows that match your query and committed.

Example: You query for all employees in a department and get 10. Another user adds a new employee to that department and commits. When you run the same query again, you get 11 rows.

To solve these more complex issues, the next topics to study are the higher isolation levels: Repeatable Read and Serializable.

Summary

The subject of today's content is read committed isolation. Let's start with the basics of databases. Databases are multi-threaded, meaning they have many reader and writer threads executing concurrently on the same computer. Because we don't know the exact order of these executions, many issues can arise that threaten the correctness of our results.

This concurrency introduces race conditions. Before we discuss examples, let's define some terminology, specifically the word 'commit'. Committing a write means the right is confirmed in the database. If a transaction involves a sequence of writes, such as to two different rows, both writes must be successful for the transaction to be committed. If only the first write has succeeded, it is not yet committed; all operations in the transaction must be finished.

Let's discuss examples of race conditions, starting with a 'dirty write'. This occurs when one transaction writes over the uncommitted values of another transaction. A dirty write can lead to an inconsistent state that breaks database invariants. Row-level locks are the solution to fix dirty rights.

However, dirty rights aren't the only issue. Another is the 'dirty read', which is reading uncommitted data. For instance, consider a transaction (T1) that deducts $10 from an account balance. A second transaction (T2) reads this new, lower balance. However, if T1 later fails and rolls back (for example, the corresponding credit to another account fails), T2 has read data that was never officially committed and is now incorrect. This is a dirty read.

How can we fix dirty reads? One option is row-level locking, but this is inefficient for reads, as locking is slow and can create bottlenecks. A better solution is for the database to maintain the old, committed value of the data. Readers will see this old value until the writing transaction successfully commits. This approach, often part of Multi-Version Concurrency Control (MVCC), incurs some storage overhead but significantly speeds up the database by reducing the need for read locks.

In summary, we've covered two types of race conditions: dirty reads and dirty rights. A database that protects against both is implementing 'read committed' isolation. It's important to note that many more race conditions exist beyond these two. Subsequent material will cover more advanced topics and the more complex solutions required to handle them.