🧠 Advanced Class Notes: Database Isolation, Concurrency Problems, Optimistic Locking, Pessimistic Locking & Redis-Based Distributed Locks

---

🎯 What You'll Learn

In this class, you'll master:

- What isolation means in databases and why it's critical.
- Different types of concurrency issues and how they impact data reliability.
- The four isolation levels in MySQL: pros, cons, and exact behavior.
- What Optimistic Locking is, how it differs from Pessimistic Locking, and when to use each.
- How Redis can be used for fast, scalable, distributed locking with real-world booking scenarios.

📘 1. What is Isolation in a Database?

🔹 Definition

Isolation ensures that multiple concurrent transactions **don't interfere** with each other. Every transaction should execute as if it were running **alone**, even if in reality, others are running at the same time.

This ensures:

- **Consistency** of reads and writes.
- No confusion in multi-user environments like online banking or flight bookings.

🔹 Part of ACID

- **Atomicity** – All or nothing.
- **Consistency** – Valid state before and after.
- **Isolation** – Appears as serial execution.
- **Durability** – Committed changes persist.

🔍 2. Why Is Isolation Important?

Let's say you're using an `accounts` table in a banking app:

```
CREATE TABLE accounts (
  id INT PRIMARY KEY,
  name VARCHAR(50),
  balance INT
);
```

Initial data:

id	name	balance
1	Sanket	1000
2	Sartak	1000

Concurrent Booking Scenario

Two users are updating the same account:

Transaction 1 (T1):

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 500 WHERE id = 1;
-- COMMIT is delayed (maybe due to long computation)
```

Transaction 2 (T2):

```
START TRANSACTION;
SELECT balance FROM accounts WHERE id = 1;
-- Returns 500
```

Now if T1 **rolls back**, T2 made decisions based on a **non-final state**. This is a violation of isolation — specifically a **dirty read**.

---

3. Types of Concurrency Issues

1. Dirty Read

Reading **uncommitted** data from another transaction.

```
-- T1:
START TRANSACTION;
UPDATE accounts SET balance = 500 WHERE id = 1;
-- COMMIT is NOT called yet

-- T2:
SELECT balance FROM accounts WHERE id = 1;  -- Gets 500 (unsafe)
```

If T1 rolls back, T2 has already used invalid data. Not acceptable in banking or e-commerce.

---

2. Non-Repeatable Read

Same transaction reads the same row twice and gets **different values**.

```
-- T1:
START TRANSACTION;
SELECT balance FROM accounts WHERE id = 1; -- 1000

-- T2:
UPDATE accounts SET balance = 900 WHERE id = 1;
COMMIT;
```

```sql
-- Back to T1:
SELECT balance FROM accounts WHERE id = 1; -- Now 900!
```

T1 observed the **world changing mid-flight** — this breaks consistency.

🔹 3. Phantom Read

Transaction reads a **range of rows** twice and gets **a different number of rows**.

```sql
-- T1:
START TRANSACTION;
SELECT * FROM accounts WHERE balance > 1000; -- 0 rows

-- T2:
INSERT INTO accounts(name, balance) VALUES ('Tanmay', 2000);
COMMIT;

-- Back to T1:
SELECT * FROM accounts WHERE balance > 1000; -- Now 1 row
```

A “phantom” row appears — hence the name.

🔹 4. Isolation Levels in MySQL

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
READ UNCOMMITTED	✅ Allowed	✅ Allowed	✅ Allowed
READ COMMITTED	❌ Avoided	✅ Allowed	✅ Allowed
REPEATABLE READ	❌ Avoided	❌ Avoided	✅ Allowed
SERIALIZABLE	❌ Avoided	❌ Avoided	❌ Avoided

🔹 Set Isolation Level

```sql
-- For the current session
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- For all sessions (global default)
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

🔹 Choosing the Right One

Use Case	Recommended Level
Analytics with no writes	READ UNCOMMITTED
Most applications	READ COMMITTED
Ecommerce / finance apps	REPEATABLE READ (default)
Payments / critical systems	SERIALIZABLE

🔒 5. Pessimistic Locking

🧠 What Is It?

Pessimistic locking assumes **conflict is likely**. When a transaction reads a row, it **locks** it so that no one else can read/write until it's done.

This avoids conflicts but reduces concurrency.

🛠 How to Implement

```
START TRANSACTION;
SELECT * FROM accounts WHERE id = 1 FOR UPDATE;
-- This locks the row. Others trying to read/write will wait.
```

Only after this transaction **commits or rolls back** will others get access to the row.

🧪 Example: Inventory System

Two customers buying the last unit of a product:

```
-- T1:
START TRANSACTION;
SELECT stock FROM products WHERE id = 99 FOR UPDATE;
-- stock = 1, proceed to buy

-- Meanwhile T2 blocks and waits until T1 finishes.
```

✅ Pros

- Extremely safe.
- Great for high-conflict environments (banking, inventory, ticket booking).

❌ Cons

- Low performance due to blocked reads/writes.
- Risk of **deadlocks**.

🔓 6. Optimistic Locking (OL)

🧠 What Is It?

Instead of locking rows, we assume there won’t be conflicts. We detect changes by checking a **version number** or **last updated timestamp** at write time.

🧪 Example with Version

```
-- Step 1: Read version
SELECT id, version FROM bookings WHERE id = 42;  -- version = 3

-- Step 2: Update
UPDATE bookings
SET status = 'confirmed', version = version + 1
WHERE id = 42 AND version = 3;
```

If the row was changed by someone else, this query fails (0 rows affected).

✅ Use OL When:

- Low probability of write conflicts
- High concurrency
- Systems like social apps, product pages, likes/upvotes

🔐 7. Redis-Based Distributed Locking

🤔 Why Not Use DB Locks?

- They’re slow
- Can deadlock
- Can’t scale across microservices

🧠 How Redis Solves This

Redis allows **fast key-based locks** with **TTL (time to live)**.

```
// Node.js Redis Example
redis.set("lock:ride:101", "user1", "NX", "EX", 30);
```

- `NX` = set only if not exists (i.e. atomic lock)
- `EX` = expire in 30 seconds

🔄 Example Use Case: Booking System

- Lock: "lock:booking:<resource_id>"
- Unlock: Either explicitly or after TTL expires
- Use retry + backoff if locked

🧰 Redis Lock Benefits

Feature	Redis Lock	DB Lock
Speed	✅ Blazing fast	❌ Slower
Deadlock Safe	✅ Yes	❌ No
Cross-System Friendly	✅ Yes	❌ No
TTL-Based Auto Cleanup	✅ Yes	❌ No

🧠 Final Summary

Strategy	When to Use	Pros	Cons
Pessimistic Locking	Bank transactions, race-prone writes	High safety	Slower, risk of deadlocks
Optimistic Locking	Rare conflicts, high performance apps	Fast, no locking	Retry logic required
Redis Locking	Distributed booking/job queues	Distributed, fast	Needs infra setup

---

What Next?

Would you like me to:

- Turn this into a downloadable PDF or Notion format?
- Add MCQs or system design problems based on this?
- Show you Java or Node.js integration for each?

Let me know how you want to study next.